VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Technologie Oracle Fusion Middleware
Technology Oracle Fusion Middleware

2012 Bc. Michal Ruman

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Michal Ruman**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Technologie Oracle Fusion Middleware
Technology Oracle Fusion Middleware

Zásady pro vypracování:

Hlavním cílem této práce je popsat a seznámit čtenáře s technologií Oracle Fusion Middleware, zejména pak s Oracle WebLogic a technologií Oracle ADF na platformě Java. Práce se zaměří na tvorbu aplikací pomocí technologie Oracle ADF v kombinaci s technologickým rámcem Spring. Nedílnou součástí pak bude představení těchto technologií na netriviální ukázkové aplikaci, která vyzdvihne a představí nejdůležitější výhody využití zvolených technologií.

Práce bude obsahovat:
1. Popis technologií Oracle Fusion Middleware, Oracle WebLogic, Oracle ADF a rámce Spring se zaměřením na vnitřní architekturu jednotlivých technologií.
2. Netriviální ukázkovou aplikaci a popis její struktury a implementace v návaznosti na představované technologie.

Seznam doporučené odborné literatury:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, Leonard L. Tripp. Guide to the Software Engineering Body of Knowledge - 2004 Version IEEE, 2004. Dostupné z www: <http://www.computer.org/portal/web/swebok/htmlformat>. ISBN 0-7695-2330-7
Oracle : Oracle Application Development Framework - Oracle ADF [online]. 2011 [cit. 2011-11-28]. Dostupné z WWW: <http://www.oracle.com/technetwork/developer-tools/adf/overview/index.html>

Dále dle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 3.5.2012 .....

.....
Bc. Michal Ruman

Poděkování

Rád bych poděkoval Ing. Davidu Ježkovi, Ph.D. za jeho cenné rady.

Abstrakt

Cílem této práce je seznámení s technologiemi, které jsou součástí Oracle Fusion Middleware, konkrétně pak s Oracle WebLogic a Oracle ADF. Je uvedena souvislost těchto technologií s principy SOA a jejich místo v architektuře SOA. Dále je vzpomenut technologický rámec Spring a možnost jeho použití při implementaci aplikace na platformě Java. Součástí je implementace ukázkové aplikace využívající rámec Oracle ADF a WebLogic. Práce se dále zabývá řešením problému pádu aplikace při nedostatku paměti, což je v souvislosti s použitými technologiemi častý problém.

Klíčová slova

Java, Oracle Fusion Middleware, Oracle WebLogic, Oracle ADF, JDeveloper, Spring, Garbage Collector, OutOfMemory.

Abstract

The main goal of this document is introducing technologies, which are part of Oracle Fusion Middleware - Oracle WebLogic and Oracle ADF especially. Relationship of these technologies in SOA principles and their place in SOA architecture are mentioned. In addition, Spring Framework and its capabilities during application development in Java platform are described. Part of this work is implementation of sample application using Oracle ADF and WebLogic. Finally, solving application crashes in consequence of memory leak is shown, because it is common problem encountered when using Java technologies.

Keywords

Java, Oracle Fusion Middleware, Oracle WebLogic, Oracle ADF, JDeveloper, Spring, Garbage Collector, OutOfMemory.

Obsah

1. Úvod

Práce se zabývá technologiemi *Oracle Fusion Middleware* a seznamuje čtenáře s možnostmi, které přináší při řešení rozsáhlých enterprise systémů. Po seznámení se s tímto širokým technologickým záběrem se práce zaměřuje na některé části *Oracle Fusion Middleware*, konkrétně *Oracle WebLogic* a *Oracle Application Development Framework*.

V části věnované *Oracle WebLogic* je představeno, co tato technologie přináší, dále je pak podrobněji představen *Oracle WebLogic Server* se zaměřením na vnitřní architekturu a rozdělení správních celků.

Část věnovaná rámci *Oracle Application Development Framework* popisuje benefity, které nám použití tohoto rámce přináší. Je představena architektura předložená tímto rámcem a vysvětleno, jak tento rámec použít.

V praktické části je využito uvedených technologií na ukázkové aplikaci. Ta využívá Aplikačního rámce *Oracle ADF*, běží na aplikačním serveru *Oracle WebLogic* a k perzistenci dat využívá relační databázi *Oracle 11g*.

Poslední část je věnována jednomu z problémů, které se často na aplikacích běžících na platformě *Java* vyskytují, jímž je *OutOfMemory*. OutOfMemory je názvem výjimky, kterou zapříčiní nevhodná konfigurace nebo *memory leak*. Jelikož je tento problém častý právě na větších Java EE aplikacích, je toto téma velice aktuální.

2. Oracle Fusion Middleware

Společnost Oracle Corporation byla založena roku 1977. Dlouhá léta byla chápána jako přední poskytovatel řešení pro perzistenci dat, konkrétně relačních databází. V posledních letech však společnost rozšiřuje své působení v oblasti ICT. Z pohledu této práce jsou velice zajímavé akvizice společností BEA Systems (duben 2008) a Sun Microsystems (leden 2010), které společnost v rámci svého rozšiřování provedla.

Akvizicí společnosti BEA Systems rozšířila své portfolio o produkty jako:

- BEA Weblogic – nyní Oracle Weblogic Server, jedna z nejpoužívanějších platforem pro Java EE infrastrukturu. Obsahuje produkty jako WebLogic Server, WebLogic Workshop, WebLogic Portal, WebLogic Integration a JRockit.
- AquaLogic – nyní Oracle Service Bus, Servisně orientovaná platforma (SOA, viz dále).
- Tuxedo – transakčně orientovaná platforma (middleware platforma).

Akvizicí společnosti Sun Microsystems získala mimo jiné následující produkty:

- Sun Servers,
- Solaris,
- Java,
- MySQL a Java DB,
- OpenOffice.org,
- GlassFish middleware řešení,
- NetBeans vývojové prostředí,
- VirtualBox – řešení virtualizace,
- a další.

Tyto produkty tedy rozšířily již existující portfolio o velice zajímavá řešení. Výše uvedené již také napovídá, že je společnost produktově velice dobře vybavena k realizaci rozsáhlých enterprise řešení.

V mnoha podnicích jsou dnes provozovány spousty různých aplikací řešících různé požadavky, problémy. S postupem času přichází požadavky na možnosti jejich vzájemného propojení, spolupráce. Často pak narážíme na problém, že jednotlivé části jsou realizovány za použití různých technologií, od různých výrobců, běžící na různých platformách apod. Řešení těchto problémů dalo vzniknout nové kategorii software – *middleware*. Co vlastně *middlerare* je? Jde o software, který slouží jako překladatelská, konverzní vrstva mezi takovýmito různorodými systémy. Je to vrstva, která nám umožní komunikaci mezi těmito nesourodými částmi kýženého celku, které by se mezi sebou jinak nedomluvily. Mnohdy se také používá termín integrační vrstva, která integruje jednotlivé kousky do jednoho funkčního celku. Existuje mnoho middleware řešení, která nám umožní propojit takové systémy. Jedná-li se o propojení 2 aplikací, nemusí jít o nic složitého. Problém nastává, chceme-li integrovat velké množství takovýchto aplikací. Zde se již nemusí jednat o jednoduchý úkol.

Jaké řešení tedy nabízí použití *Oracle middleware*? Nabízí nám, že implementujeme-li jednotlivá řešení pomocí Oracle technologií, nebude pak (velký) problém jednotlivé části vzájemně propojit i při jejich větším množství. Řešíme-li infrastrukturu pomocí tohoto řešení, umožňuje nám

vytvářet a používat výkonné a inteligentní obchodní aplikace, které budou vzájemně využívat svůj potenciál. Oracle shrnul popis do několika bodů:

- Úplnost — Řešte všechny své požadavky na middleware s jediným strategickým partnerem.
- Integrace — Certifikované integrace s produkty Oracle Fusion Middleware, Oracle Database a Oracle Applications zaručují spolehlivost a snižují náklady.
- Možnost připojení za provozu — Rozšiřujte svou stávající infrastrukturu a aplikace bezkonkurenčně snadno a operativně.
- Nejvyšší kvalita — Vybírejte z nejlepších řešení svého druhu ve všech produktových řadách. [10]

Jak bylo nastíněno výše, díky produktům, které Oracle vyvinul sám, nebo je získal díky strategickým akvizicím, může skutečně pokrýt širokou škálu požadavků. Tak je možné mnoho různých požadavků řešit pouze za pomoci Oracle technologií, což s sebou může přinést snížení celkových nákladů (na každou další technologii nižší cena). Taková řešení jsou pak také snadněji integrovatelná, než kdyby se jednalo o řešení částečně realizovaná na jiných platformách. Tyto integrace se pak mohou provádět i postupně, což je nespornou výhodou.

Hovoříme-li o těchto technologiích, nemáme na mysli pouze middleware, ale celou spoustu nejrůznějších řešení. Oracle toto souhrnně označuje jako Oracle Fusion Middleware, základ pro celou infrastrukturu. Jaké části tedy Oracle Fusion Middleware zahrnuje?

2.1 Části Oracle Fusion Middleware

Jednotlivé části jsou přehledně shrnuty v následující tabulce. Samotnou kapitolou jsou vývojové nástroje, které stojí vedle dalších částí a poskytují pro ně podporu. Stejně je tomu tak u řešení Enterprise Managementu a Identity Managementu.

Vývojové nástroje	User Interaction Enterprise Performance Management Business Intelligence Content Management SOA &Process Management Application Server Grid Infrastructure	Enterprise Management Identity Management

2.1.1 Vývojové nástroje

Samostatnou kapitolou jsou vývojové nástroje, které jsou součástí Fusion Middleware. Umožňují jednotný vývoj SOA aplikací, poskytují sadu integrovaných nástrojů a rámců pro vývoj aplikací, databází a business inteligence. Nabízí také nástroje a rozšíření pro produkty jako Oracle JDeveloper a Eclipse.

Nabídka nástrojů je poměrně široká:

Java a SOA

- Oracle JDeveloper
- Vývojářská sada Oracle pro Spring
- Oracle Enterprise Pack for Eclipse
- Oracle TopLink
- NetBeans IDE
- JavaFX
- Hudson

Solaris a Linux

- Oracle Solaris Studio
- Sada nástrojů pro předávání zpráv Oracle Message Passing Toolkit

.NET

- Oracle Developer Tools for Visual Studio
- Poskytovatel dat Oracle Data Provider for .NET
- Databázová rozšíření Oracle Database Extensions for .NET
- Producent WSRP WebCenter pro .NET

Související řešení

- Oracle TopLink
- OracleADF

Business inteligence (BI)

- BI Publisher
- Oracle BI Standard Edition One
- Oracle Data Integrator
- Oracle Reports

Databáze a PL/SQL

- Application Express
- Formuláře (Oracle Forms)
- SQL Developer

Rozhraní JAVA API

- Rozhraní Java TV API
- Rozhraní Java Card API

Balíky Java SDK

- Platforma Java SE (JDK)
- Platforma Java ME (SDK)
- Platforma Java EE (SDK)

2.1.2 User Interaction

Web 2.0 Portal, Rich Internet Apps, Mobile, Search, Desktop, Presence, VoIP

2.1.3 Enterprise Performance Management

Plánování, Rozpočtování, Řízení financí a vykazování, Ukazatele výkonnosti (tzv. Scorecards)

2.1.4 Business Intelligence

Integrace dat, Dotazy a analýzy, OLAP (multidimensional analytic engine), Dashboards, Reporty, Upozornění, Real-Time.

2.1.5 Content Management

Webový obsah, Dokumenty, Records Management, DAM (Digital asset management), Capture and Imaging, Archivování, IRM (Information Rights Management).

2.1.6 SOA & Process Management

ESB, BPM, Workflow, BAM, Rules, B2B, MDM, Registry, Repository.

2.1.7 Aplikační servery

Java EE, Webové služby, TP monitor, Komplexní zpracování událostí (Complex event processing), XTP (Extreme Transaction Processing), SIP (Session Initiation Protocol – protokol pro inicializaci relací).

2.1.8 Grid Infrastructure

Aplikační Cluster, In-Memory data grid, Metadata Services, JVM, Virtualizace.

2.1.9 Enterprise Management

SOA Management, Provisioning, Diagnostika, Konfigurační management, Ladění (Tuning).

2.1.10 Správa identit

Provisioning, Management přístupů (Access Management), Audit, Directory, Management rolí (Role Management), Detekce podvodů (Fraud management).

2.2 Middleware propojitelný za provozu s dalšími systémy

Jak již bylo uvedeno, jednou z vlastností infrastruktury realizované na Fusion Middleware technologiích je možnost jednoduchého připojení k dalším systémům, jednoduché rozšíření stávající infrastruktury o nové části. Velkou výhodou je, že se nemusí jednat jen o řešení z nabídky Oracle, ale je možné využít i dalších řešení. Opět bude nejpřehlednější rozdělit si tyto do výše definovaných kategorií.

2.2.1 Vývojové nástroje

Při vývoji nemusíme využít jen toho, co nabízí Oracle, ale můžeme použít spoustu dalších oblíbených nástrojů.
Mezi nejvýznamnější patří:

Eclipse, CollabNet Subversion, Spring, Struts, JUnit, Ant, Tapestry, SVN, CVS, MS Visual SourceSafe.

2.2.2 User Interaction

WSRP a JSR-168 portály, MS Office, Wireless a mobilní zařízení.

2.2.3 Enterprise Performance Management

SAP ERP & BW, Excel, Outlook, Teradata, DB2.

2.2.4 Business Intelligence

Teradata, DB2, MS Analysis Services, SAP BW, Cognos, Business Objects.

2.2.5 Content Management

Microsoft Office, Adobe PDF, Microsoft SharePoint.

2.2.6 SOA & Process Management

IBM WebSphereMQ, TIBCO Enterprise, Message Service, SonicMQ.

2.2.7 Aplikační servery

BEA WebLogic, IBM WebSphere, JBoss App Server, Apache Tomcat.

2.2.8 Grid Infrastructure

Certifikováno na všech hlavních operačních systémech, Common Metadata Services.

2.2.9 Enterprise Management

HP OpenView, CA Unicenter, IBM Tivoli, BMC Patrol.

2.2.10 Správa identit

MS Active Directory & MIIS, CA eTrust SSO, všechny LDAP adresáře.

2.2.11 Aplikace a databáze

Vedle uvedených je dále možné integrovat aplikace jako je SAP R/3, mySAP, všechny Oracle aplikace. Stejně je tomu u databází, kde je možné využít také IBM DB2 a Informix, MS SQL Server, Sybase IQ.

2.3 SOA

V souvislosti s Fusion Middleware se často hovoří o SOA, o tom jak nám Fusion Middleware umožňuje posunout Enterprise architekturu směrem k SOA a že se jedná o komplexní platformu pro vývoj a nasazení servisně orientovaných aplikací.

Co tedy SOA je? Servisně orientovaná architektura SOA (Service Oriented Architecture) je sada přístupů a metodik pro návrh a vývoj software. Nejedná se tedy o konkrétní produkt, ale o přístup k organizování IT zdrojů s cílem maximalizovat flexibilitu managementu v podniku. Servisní architektura modularizuje IT zdroje a vytváří volně vázané business procesy určené k šířené informací v systému. Pro dobře navrženou servisně orientovanou architekturu je nezbytná nezávislost business procesů na platformě, protože pouze takto lze zajistit potřebné navýšení flexibility v podniku.

Podniková IT oddělení spravují stále komplexnější systémy. Tyto systémy musí být neustále udržovány v souladu s podnikovými cíli. Problémem pro IT oddělení není nedostatek požadované funkcionality, problém spočívá spíše v oddělení jednotlivých systémů, například *customer relationship management* (CRM) a *enterprise resource planning* (ERP), které pracují samostatně a sloučení informací poskytovaných těmito systémy je velmi problematické. Dříve byla často uplatňována časově náročná manuální řešení nebo byl problém řešen díky speciálním pomocným programům, které však snižovaly flexibilitu a udržovatelnost IT systémů.

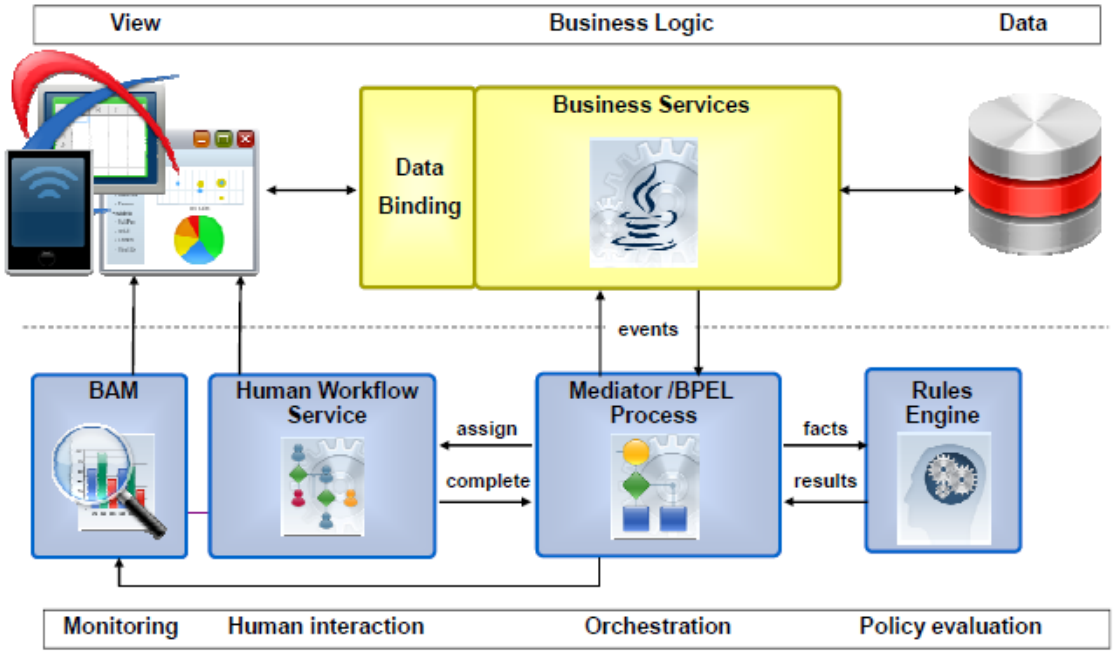
Servisně orientovaná architektura přináší řešení ve formě vysoce flexibilních dynamických aplikací (někdy také nazývaných kompozitní). Tyto aplikace dovolují získávat informace mnohem snadněji a v mnohem přístupnější podobě. Koncový uživatel si může zvolit nejen formu výstupních dat, ale můžu tato data zpracovávat a prohlížet na celé škále zařízení, ať už jde o webový portál, aplikačního klienta nebo mobilní přístroj. Dynamické aplikace dovolují podnikům zlepšovat a automatizovat manuální úlohy, zpřehledňují interakce se zákazníky a obchodními partnery a pomáhají lépe organizovat business procesy. Výsledkem je zvýšená konkurenceschopnost podniku na trhu.

Servisní orientace je prostředkem pro integraci rozdílných systémů. Každý IT prostředek, systém, aplikace nebo obchodní partner může vystupovat jako služba. SOA je v podstatě kolekcí služeb, které komunikují mezi sebou a ke komunikaci využívají standardizované protokoly a dohodnutá rozhraní. Díky těmto rozhraním se může měnit implementace služeb, aniž by byla ovlivněna schopnost systému služby používat. [15]

2.4 Architektura Oracle Fusion

Hlavní myšlenkou Oracle Fusion je nestavět systém jako jeden dodaný balík, ale jako sestavu různých aplikací řešících aktuálně požadované business problémy a požadavky, přičemž tyto aplikace mohou vzájemně dle potřeby spolupracovat díky dodržování filozofie SOA a využití technologií Oracle Fusion.

Jelikož zde ve velké míře hovoříme o podnikových procesech, je vhodné také uvézt, jak se tyto převádějí do praxe a jak jsou propojeny. V této souvislosti je potřeba zmínit jazyk BPEL (Business Process Execution Language), který slouží právě k modelování systémů v servisně orientované architektuře. Pomocí tohoto jazyka můžeme jednoduše modelovat business procesy. BPEL proces pak může působit na ostatní procesy, nebo jimi může být ovlivňován. Na obrázku níže je možné vidět jeho místo v architektuře Oracle Fusion.



Obrázek 1 - Architektura Oracle Fusion, zdroj: [9]

3. Oracle WebLogic

WebLogic je jedním z produktů, které Oracle získal akvizicí společnosti BEA Systems. Pod tímto pojmem je znám hlavně jako aplikační server, ale není jen tím. Aktuálně Oracle WebLogic zahrnuje několik částí:

- WebLogic Server – známý aplikační server oblíbený pro svou robustnost, širokou nativní podporu standardů.
- JRockit – vlastní Java Virtual Machine, která by měla poskytovat lepší výkon při provozování EE aplikací.
- WebLogic Portal – portálový rámec pro vytváření vysoce interaktivních skládaných aplikací v prostředí SOA. Integrované vývojové prostředí pro vývoj portálových řešení.
- WebLogic Workshop – vlastní vývojové prostředí (IDE) pro vývoj Java aplikací, webových aplikací, Portálových aplikací, webových služeb, SAO, Rich internet aplikací apod. Dříve samostatná aplikace, od verze 9.0 se již jedná o nástavbu nad prostředím Eclipse.

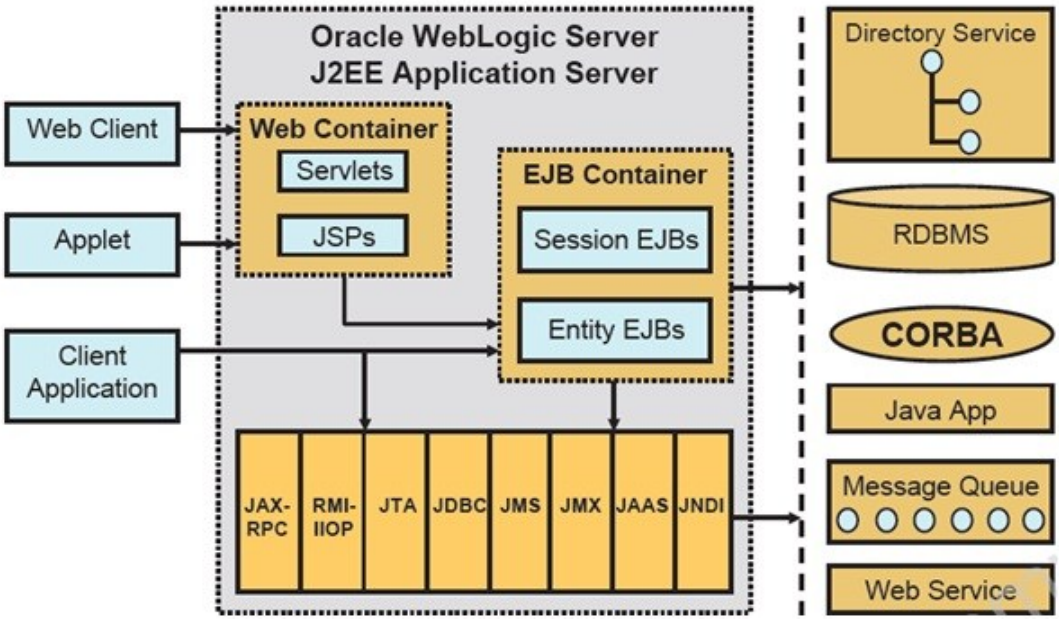
3.1 Oracle WebLogic Server

Zaměříme se nyní na Oracle WebLogic Server a popíšeme si, jaké možnosti nám přináší. Ačkoli byla v prosinci 2011 vydána nová verze WebLogic Server 12c a většina dále uvedených věcí by měla platit i pro ni, budeme se opírat o vlastnosti verzí 9, 10 a 11g (11gR1).

WebLogic Server je aplikační server založený na J2EE standardech pro distribuované systémy. Distribuované systémy jsou takové, které dělí práci mezi několik nezávislých modulů. Cílem je minimalizovat při výpadku jednoho modulu dopady na zbytek systému, čímž je docíleno vyšší dostupnosti, škálovatelnosti a udržovatelnosti. Mnohé výhody distribuovaných systémů vychází z využívání standardů jako:

- zajištění rozdělení složitých problémů na oddělené platformy
- umožnění modularizace komplexního hardware a software
- umožnění vynaložit větších části projektových nákladů na řešení obchodních softwarových potřeb

WebLogic Server je tedy postaven na J2EE Architektuře. Na následujícím obrázku je popis J2EE architektury definované společností Sun Microsystems a aplikovaný ve WebLogic Serveru. Každý J2EE aplikační server musí být v souladu s touto architekturou. Mnohé servery nabízí i další služby, ale nikdy by se v základních aspektech neměly lišit od popsané architektury.



Obrázek 2 - J2EE Architektura, zdroj: [13]

3.1.1 Architektura Oracle WebLogic Serveru

Jak vidíme na obrázku výše, WebLogic Server se skládá z několika částí. Popíšeme si tedy nejdůležitější.

3.1.1.1. Webový kontejner

Webový kontejner (Web Container) poskytuje aplikační prostředí pro Webové aplikace, poskytuje infrastrukturu, uvnitř které Webové aplikace běží. Požadavky na infrastrukturu definuje Servlet specifikace (WLS 9.0 – Servlet 2.3, WLS 12c – Servlet 3.0, ….). Webový kontejner zajišťuje veškeré interakce jako je nasazení (*deploy*), běh a přístup k webovým aplikacím. Kontejner také musí poskytnout služby jako je serverové API, podporu a služby v době běhu pro vytváření dynamického obsahu a poskytování statického obsahu, dále pak podporu pro nasazení aplikace.

Přijde-li požadavek z klienta (GET, POST), přichází na Webový kontejner, který je na základě konfigurace mapován na konkrétní webovou aplikaci. Takzvaný *deployment descriptor* (`web.xml` a `weblogic.xml`) definuje, jak jsou URL mapovány na konkrétní webovou aplikaci. Webové aplikaci přijde požadavek a ta může klientovi odpovědět.

3.1.1.2. EJB kontejner

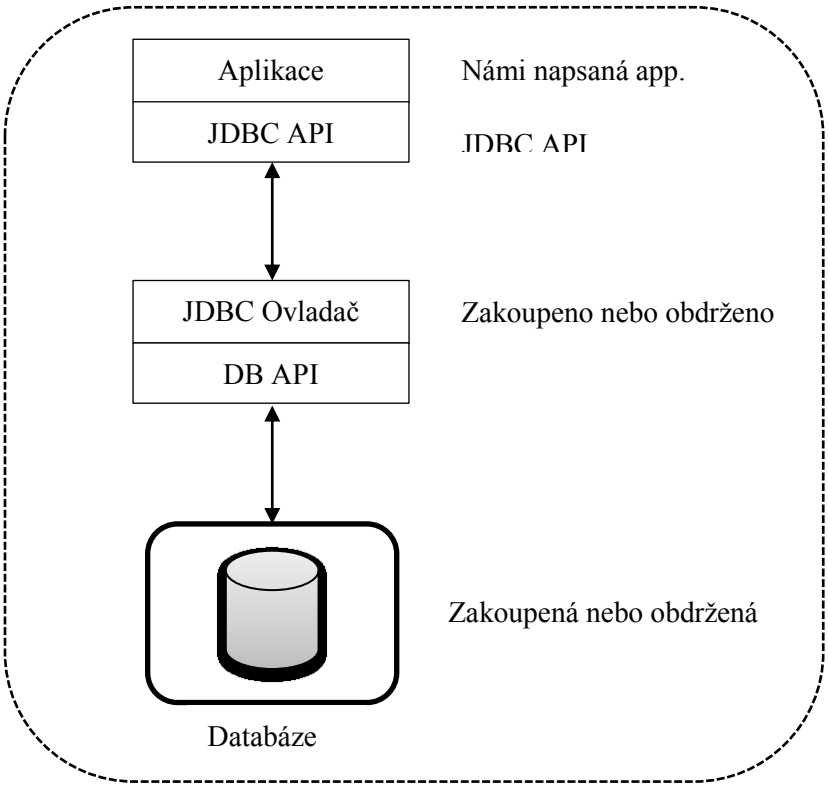
Podobně jako Webový kontejner, také EJB kontejner vytváří běhové prostředí, tentokrát pro Enterprise JavaBeans. Jak plyne ze specifikace, řídí jejich životní cyklus, poskytuje jim přístup

ke zdrojům (JDBC apod.), stará se o autentizaci a autorizaci a poskytuje další služby. Řídí také transakce a umožňuje distribuci komponent prostřednictvím RMI-IOOP. Nastavení těchto služeb je řešeno v souborech deployment descriptor, nebo pomocí anotací.

Podporovány jsou Stateless Session EJB, Statefull Session EJB, Message-driven EJB, Singleton EJB a Entity EJB.

3.1.1.3. Java Database Connectivity (JDBC)

WebLogic poskytuje standartní Java rozhraní pro zpřístupnění heterogenních databází. Na následujícím obrázku jde vidět, jakým způsobem je JDBC používáno.



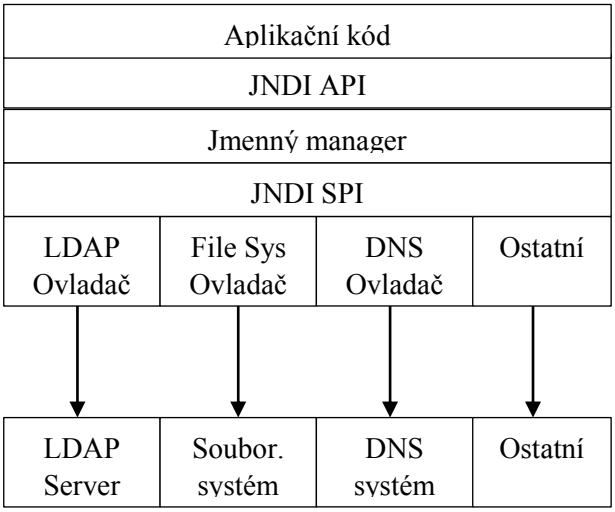
Obrázek 3 - Schéma JDBC, zdroj: [8]

Specifikace definuje čtyři typy ovladačů pro připojení k databázi:

- *Typ 1* využívá k přístupu k databázi JDBC-ODBC most, takže k databázi vlastně přistupuje přes ODBC.
- *Typ 2* využívá nativního ovladače nainstalovaného v počítači a překládá JDBC požadavky do podoby srozumitelné tomuto nativnímu ovladači.
- *Typ 3* je čistě Java ovladač, který převádí svou komunikaci do síťového protokolu srozumitelnému databázi.
- *Typ 4* je ovladač napsaný v Javě a komunikuje přímo s databází, tedy převádí volání přímo do protokolu databáze bez dalšího prostředníka.

3.1.1.4. Java Naming &Directory Interface (JNDI)

Java API pro zpřístupnění jmenných a adresářových serverů. Jde o vrstvu nad DNS, LDAP atd.



Obrázek 5 - Architektura při použití JNDI, zdroj: [8]

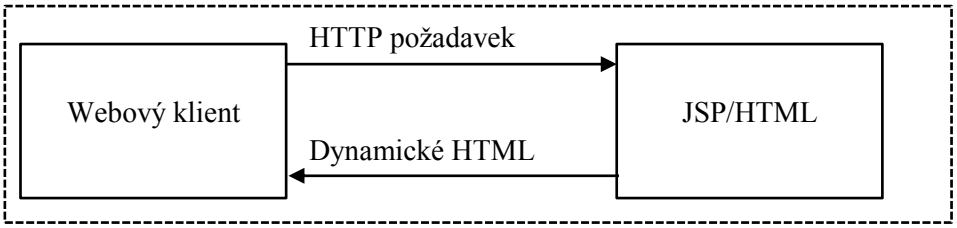
3.1.1.5. Java Servlets

Servlet je Java třída poskytující službu na základě protokolu request/response. Technicky vzato, servlet je jakákoli Java třída implementující rozhraní `javax.servlet.Servlet`.
Může běžet na mnoha různých heterogenních prostředích a být vyvolán přes různé protokoly. Nejčastěji se setkáme se servletem v podobě http Servletu, který je vyvolán standartním HTTP paketem zaslaným z klientského prohlížeče.

WebLogic Server servlet má automaticky generováno SessionId v cookies a URL. Jsou nasazovány uvnitř webových aplikací.

3.1.1.6. Java Server Pages (JSPs)

JSP soubory jsou textové dokumenty, které vytváří dynamické webové stránky na základě požadavků klienta. Jsou překládány na servlety. Obsahují kombinaci HTML značek, JSP značek a Java kódu.



Obrázek 6 - JSP stránka, zdroj: [8]

3.1.1.7. Java Transaction API (JTA)

JTA je standartní Java rozhraní sloužící k vymezení transakcí uvnitř programu. WebLogic podporuje lokální a distribuované transakce.

3.1.1.8. Java Message Service (JMS)

Java rozhraní pro zpřístupnění middleware založeného na posílání zpráv. Rozhraní podporuje Garantované doručení zprávy, transakce a další. Jedná se o mocný nástroj sloužící k asynchronní výměně dat a událostí.

3.1.1.9. Enterprise JavaBeans (EJBs)

Jedná se o distribuované komponenty napsané v jazyce Java. Poskytují klientovi distribuovatelné a nasaditelné business služby. Mají definované rozhraní a jsou tak znovupoužitelné meze aplikačními servery. Běží v kontejneru, který jim poskytuje správu a kontrolní služby.

3.1.1.10. Java Authentication and Authorization (JAAS)

Jedná se o bezpečnostní rámec podporující single sign-on (jednotné přihlášení napříč heterogenními systémy s mnoha autentikačními systémy) a připojitelný autentikační modul (PAM). Umožňuje flexibilní kontrolu nad autorizací založenou na *uživatelích, skupinách* a *rolích*.

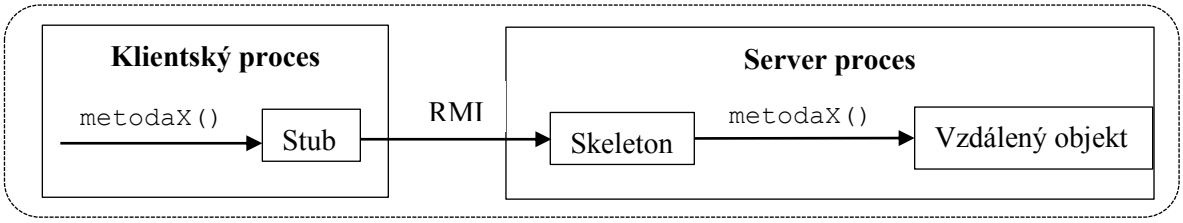
3.1.1.11. Java Management Extensions (JMX)

Definuje a standardizuje infrastrukturu pro správu zařízení z Java aplikací. Odděluje také spravovaná zařízení od nástrojů pro jejich správu.

3.1.1.12. Remote Method Invocation (RMI)

RMI, neboli vzdálené volání služeb umožňuje objektům vyvolat metodu ve vzdáleném objektu použitím *stubs* a *skeletons* (použití překladu *pahýl* a *kostra* je v tomto místě nevhodné,

originální slova jsou zažité termíny). RMI používá serializaci k předání dat hodnotami mezi dvěma objekty.



Obrázek 7 - Schéma vzdáleného volání metody, zdroj: [8]

3.1.1.13. Web Services

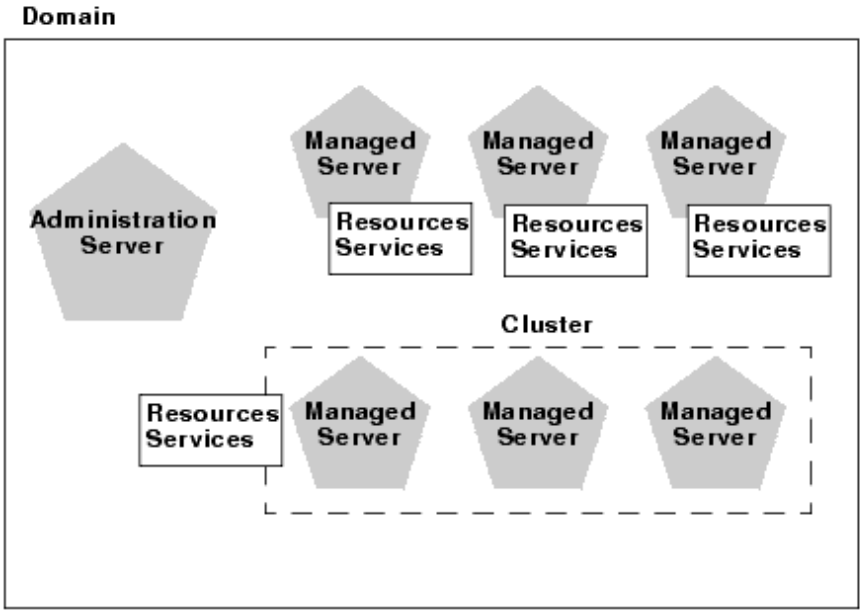
Webové služby. Jedná se o služby na straně serveru přístupné prostřednictvím webových protokolů. K jejich popisu se používá XML a prostřednictvím XML také komunikují. Jsou jedním z vývojových paradigmat umožňujících posunutí naší architektury směrem k SOA. Umožňují komunikaci mezi různými systémy na různých platformách, například Java aplikační server hostí webovou službu, kterou využívá klient běžící na MS serveru napsaný v C#.

3.1.2 Základní správní jednotky WebLogic serveru

Pro práci s WebLogic serverem je důležité znát koncept, jakým způsobem server pracuje se spravovaným obsahem a jaké definuje správní celky. Tyto celky definujeme po instalaci WebLogic serveru.

3.1.2.1. Doména

Na úplném vrcholu hierarchie se nachází doména. Jedná se o jednoduchou administrační jednotku, v jejímž rozsahu administrační server vidí spravované *managed* servery. Doména obsahuje jednotlivé servery nebo clustery serverů. Všechny informace o doméně jsou v konfiguračních souborech, neexistují WLS programová rozhraní odkazující se na doménu, jde čistě o organizační jednotku. Každá doména obsahuje jeden Administrační server a 0 – n *managed* serverů, viz dále. Můžeme použít jednu instalaci WLS k vytvoření a provozování několika domén, nebo můžeme použít několik instalací k provozování jediné domény. To závisí na našich potřebách.



Obrázek 8 - Schéma domény, zdroj: [14]

3.1.2.2. Server

Server je ve své podstatě instance třídy `weblogic.Server` běžící uvnitř JVM. Server může být asociován s maximálně jedním WLS strojem, ačkoli stroj může obsahovat více serverů. Server má vyhrazenu vlastní část paměti a je více-vláknový.

3.1.2.3. Administrační Server

Administration Server (dále jen admin server) funguje jako centrální řídící bod pro konfiguraci celé domény. Drží u sebe XML konfigurační repository (config.xml). Admin server musí běžet, pokud chceme provézt jakékoli změny ve správě domény. Na něm běží administrační konzole (dostupná na adrese serveru, například http://host:port/console). V doméně existuje vždy jen jeden admin server. Ostatní servery nazýváme *managed servery*. Každý managed server obdrží svou konfiguraci při startu od admin serveru.

3.1.2.4. Managed (spravovaný) Server

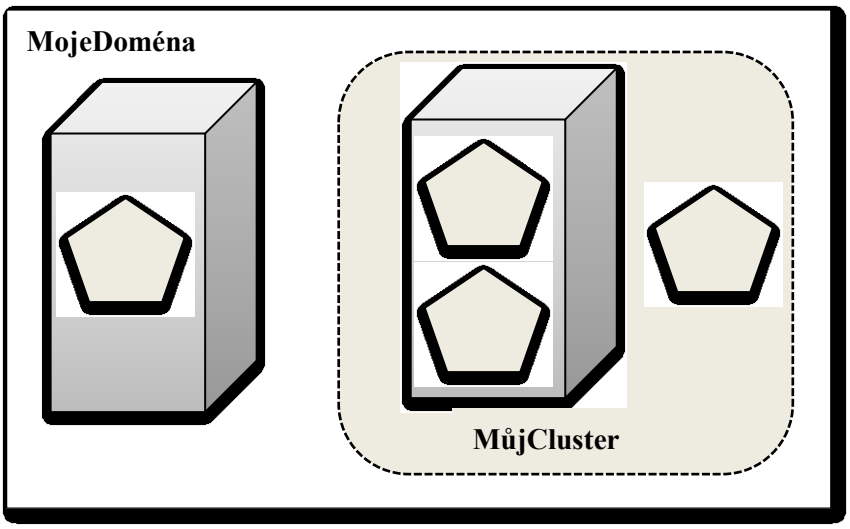
Managed server je opět instancí Weblogic serveru. Načítá svou konfiguraci vzdáleně z admin serveru. Může být, ale také nemusí, částí clusteru. V praxi je většinou admin server používán pouze na administraci v rámci domény a na managed serverech běží nasazené aplikace.

3.1.2.5. Machines (stroje)

Typicky reprezentují fyzický kus hardware, kde server spočívá. Na jednom stroji může být jeden nebo i více serverů. WLS dělá rozdíl mezi UNIX systémy a non-UNIX systémy (Například Windows, OS-X, …).

3.1.2.6. WebLogic Server Cluster

WebLogic Cluster (dále jen WLS Cluster, nebo jen cluster) může shlukovat stroje a servery. Používá se k rozdělení zátěže mezi více serverů (takzvaný *load balancing*) a jako prevence proti výpadkům dostupnosti. Každý WLS Cluster je spravován právě jedním admin serverem. Může existovat na jednom nebo více počítačích. Cluster poskytuje zapouzdřené rozhraní ke službám clusteru. Na venek se cluster klientům jeví jako jedna instance. Replikací služeb mezi více serverů v clusteru zajistíme ochranu proti výpadkům a také větší škálovatelnost. V případě potřeby například můžeme postupně restartovat servery, aniž by to mělo vliv na práci klienta/uživatele.



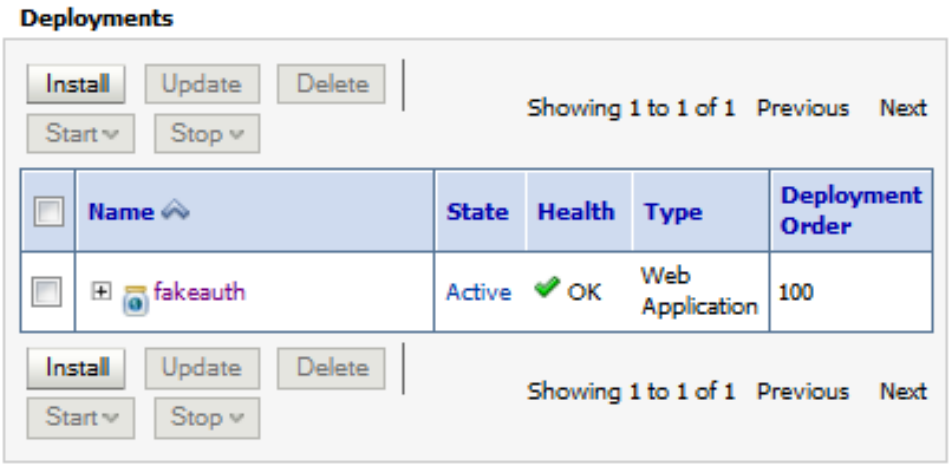
Obrázek 9 - Cluster v doméně, zdroj: [8]

Na obrázku výše je zobrazena doména s clusterem obsahujícím tři servery. Dva z nich běží na stejném stroji, takže jsou přiřazeny definici stroje. Třetí běží na samostatném stroji a my nemusíme definovat stroj, protože WebLogic Server prostě předpokládá, že je na jiném stroj. Výše uvedený cluster se třemi servery na dvou strojích je v praxi velice častá konfigurace, kdy na jednom stroji konfigurujeme admin server a managed server, na druhém stroji pak druhý managed server.

3.1.3 Konfigurace WebLogic serveru

Jak již bylo uvedeno v souvislosti se základním popisem Administračního serveru, nabízí nám WebLogic Server komfortní způsob konfigurace celé domény i jednotlivých serverů. Tímto způsobem je využití administrační konzole běžící na adrese protokol:host:port/console. Přihlašovací jméno a heslo je nastaveno při instalaci, v dřívějších verzích byla defaultní hodnota weblogic/weblogic. Ve verzích 9.0 a vyšších je však potřeba nastavit heslo jiné než jméno. Konfigurace je uložena v konfiguračních XML souborech na souborovém systému, kde je nainstalována doména, takže není problém editovat je i přímo, ale není to doporučený způsob. Na obrázku v příloze [A] je zobrazen vzhled administrační konzole po přihlášení, v tomto případě verze 11g. Konzole je přehledně rozdělena do několika kategorií a nabízí mimo možností konfigurace také diagnostické nástroje a nástroje pro sledování běžících systémů.

Co tedy WebLogic Server umožňuje? Především konfigurovat dříve uvedené jednotky, jako jsou servery, clustery a podobně. Dále nabízí správu nasazení aplikací, ať už se jedná o EAR, jednoduchou webovou aplikaci zabalenou jako WAR, nebo rozbalenou aplikaci. Na následujícím obrázku je ukázka správy nasazení – Deployments, aktuálně s nasazenou jednou webovou aplikací.



Obrázek 10 - Deployment

Při nasazení aplikace můžeme zvolit, na které servery má být aplikace nasazena a kde ne, což je z různých důvodů velice užitečné. Velice užitečným nástrojem může být také monitoring aplikací, kde můžeme sledovat počet servletů, za běhu počet aktivních session, ale i detailnější informace session, stav aplikace, klienty webových služeb apod.

3.1.3.1. JDBC Datasource

V sekci služeb je spousta možností vyplývajících z J2EE specifikace, jako je například konfigurace JMS serverů, mostů, JTA apod. Patrně nejčastěji využijeme možnost konfigurace Datasource. JDBC datasource je objekt JNDI stromu poskytující databázové spojení přes takzvané connection pools. Aplikace může z JNDI stromu získat datasource a prostřednictvím něj se pak připojit k databázi. Z toho tedy vyplývá, že každý JDBC datasource má své JNDI jméno, prostřednictvím kterého se na něj aplikace odkazuje. V rámci datasource definujeme již zmíněný connection pool. Zde se teprve odkazujeme na konkrétní databázi, definujeme použitý JDBC ovladač, přístupové údaje, ale také výkonové záležitosti, jako je inicializační a maximální počet připojení udržovaných v poolu. Možností nastavení je daleko více a jsou závislé na konkrétním prostředí, kde má být datasource využíván. Na základě konkrétních požadavků například produkčního prostředí pak nastavujeme tyto a další parametry. Jednou z výhod tohoto použití je jednoduchá zaměnitelnost cílové databáze bez změny v aplikaci, která se odkazuje na zdroj dat pomocí jeho jména v JNDI stromu. Opět máme možnost definovat, na kterých serverech bude datasource nastaven a na kterých ne.

3.1.3.2. Execution Queues & Work Managers

Jednou z věcí, o které se dozvíte až v okamžiku, kdy řešíte výkonnostní problémy, je možnost definovat takzvané Execution Queues (WebLogic 8), případně Work Managers (WebLogic 9 a vyšší). Můžeme tím definovat, které části aplikace nebo aplikací jsou pro nás stěžejní a měly by být upřednostněny i při vyšším vytížení serveru. Pro bližší informace doporučuji dokumentaci WebLogic Serveru [17], resp. jinou verzi dokumentace v závislosti na verzi WebLogic.

3.1.3.3. Security Realms

Další důležitou částí je nastavení bezpečnostních mechanismů, včetně nastavení uživatelů, skupin, uživatelských rolí, bezpečnostních pravidel a bezpečnostních poskytovatelů používaných pro zabezpečení zdrojů.

V části Security realms můžeme právě toto nastavit. Vytvoříme nový Realm a v něm nastavíme požadovanou konfiguraci výše uvedeného. Defaultně máme vytvořen realm s uživateli, jako je weblogic. Proti tomuto realmu vystupujeme, pokud se přihlašujeme do administrační konzole.

3.1.3.4. WLST

Velice často se stává, že nechceme vše ručně nastavovat pomocí administrační konzole, ale chceme proces nastavení automatizovat, například pro instalaci na nové prostředí. K tomu nám pomůže WebLogic Scripting Tool. Jedná se skriptovací prostředí na příkazové řádce umožňující vytvoření, správu a monitoring domény. Nástroj je na základě Java skriptovacího jazyka Jython, z nějž má i syntaxi, ale na pozadí využívá Java knihovny.

WLST může běžet ve dvou módech – offline a online. Rozdíl je jednoduchý, offline mód ke své práci nepotřebuje mít spuštěný server, kdežto online ano. Některé věci je lepší provádět v offline módu, po provedení některých konfiguračních změn v online módu je stejně potřeba server restartovat.

Následuje ukázka části skriptu v online módu, který vytváří skupiny a spravuje uživatele:

```
################
# Creating Groups
################

cd('/SecurityConfiguration/@domain.name@/Realms/myrealm/Authentica
tionProviders/SQLAuthenticator')
cmo.createGroup('xPortalWsUsers', 'Portal WebService users')
cmo.createUser('@admin.username@','@admin.pass@','Portal admin')
cmo.createUser('InternalAdmin','pass','Portal user administrator')
cmo.addMemberToGroup('Administrators','weblogic')
...
```

4. Oracle ADF

ADF je zkratkou pro Oracle Application Development Framework. Tak jako u mnohých jiných rámců, také ADF Framework vznikl s cílem usnadnit vývoj Java Enterprise aplikací. Ve srovnání s některými jinými je v této snaze o krok dále, hlavně ve snaze minimalizovat množství programového kódu, který musí vývojář napsat, aby vytvořil celou infrastrukturu Java EE aplikace. Aplikace umožňuje tvořit vizuálně a deklarativně, takže místo psaní kódu do aplikace vkládáme myší komponenty z dostupné palety. Ty se pak konfigurují buď v GUI vývojového prostředí, nebo editováním XML souborů. Z toho tedy vyplývá, že ADF Framework nabízí ve srovnání s jinými rámci vyšší míru abstrakce, méně kódování, zkrácení času na vývoj a tím pádem potenciálního snížení nákladů na výsledné softwarové dílo. Jakým způsobem je toho docíleno a jak ADF pracuje, si popíšeme dále. Nejdříve však bude vhodné připomenout si JavaServer Faces Framework, ze kterého ADF Framework vychází a který dále rozšiřuje.

4.1 JSF Framework

JavaServer Faces Framework (dále jen *JSF*) vyvinutý pod *JSR 314* je webový rámec založený na komponentách (*component base*) určený pro tvorbu aplikačních rozhraní, tedy zaměřený na prezenční vrstvu (na rozdíl od *Apache Struts*, který je aplikačním rámcem). V dnešní době se jedná o velice oblíbený rámec pro svou přehlednost a také množství existujících komponent různých výrobců, které můžeme využít. Co se komponent týká, nutno vyzdvihnout zvláště *RichFaces* pro jejich kvalitní zpracování, dále pak zdařilé *OpenFaces* atd. Ačkoli je již dostupná verze JSF 2.0, rozšířenější je zatím stále ještě verze JSF 1.2.

4.1.1 JSF životní cyklus

Chceme-li používat JSF, měli bychom vědět, jak pracují, jak jsou jednotlivé požadavky zpracovávány. To nám umožní pochopit a vyřešit mnohé menší i větší problémy, na které dříve či později narazíme.

Životní cyklus JSP je posloupností několika etap, kterými aplikační požadavek prochází, než je výsledek vrácen/zobrazen žadateli. Můžeme jej sledovat a zachytit ve třídě *PhaseListener*. Cyklus má 6 fází jdoucích v daném pořadí po sobě.

JSF Request lifecycle	
Restore View	Požadavek přijde skrze JSF servlet, je z něj načteno viewId svázané se stránkou. Přes něj Framework přistupuje ke komponentám stránky.
Apply Request Values	JSF komponenty jsou obdrženy z JSF kontextu. Hodnoty komponent jsou aktualizovány hodnotami z parametru požadavku.
Process Validations	Hodnoty z parametru požadavku jsou převedeny na očekávané datové typy. Provede se validace každé hodnoty komponent.
Update Model	Nyní je model aktualizován validovanými hodnotami. Modelem mohou být Managed Beans nebo ADF binding container.
Invoke Application	Dochází k zavolání aplikační logiky. Navigační pravidla jsou vyhodnocena a zpracována
Render Response	Příprava výsledků pro zobrazení.

4.1.2 Návrhový vzor Model-View-Controller (MVC)

Model-View-Controller (MVC) je jedním z návrhových vzorů (resp. skupinou vzorů – dle GoF). V ADF je tento vzor důsledně dodržován, vyplývá z něj i samotná architektura ADF, proto je vhodné říct si o něm něco více.

Čeho docílíme využíváním tohoto návrhového vzoru? Oddělíme tím část programu mající na starosti předzpracování příkazů uživatele (tj. zjištění, co od programu vlastně chce) od částí zabezpečujících logiku programu, která uživatelovy příkazy zpracovává, a části, která má na starosti zobrazení výsledků. Jinými slovy pomocí MVC lze oddělit logiku přístupu k datům od prezentace dat. Máme například stejná data, ale jiné přístupy (mobilní telefon, Internet). Do části zabývající se prezentací výsledků nepatří jen vlastní GUI, ale často také kód, který data pro tuto prezentaci připravuje. Aplikace vzoru umožňuje snadnou implementaci zadávání požadavků různými způsoby (klávesnice, myš, …), stejně jako různé možnosti prezentace výsledků (tabulka, graf, ...). Další výhodou takového rozdělení je usnadnění případných budoucích změn. Aplikace vzoru usnadňuje přenositelnost mezi platformami.

Jednoduše můžeme návrhový vzor MVC popsat následovně:
Model – tvoří jej objekty nesoucí data, která budou zobrazena prostřednictvím pohledu
Pohled (view) – přijímá od pohledu model a zobrazuje jej, zasílá odpověď
Kontrolér (controller) – zpracovává požadavek a na základě údajů, které z něj získá, vytváří model, který následně posílá pohledu.

4.2 Architektura ADF

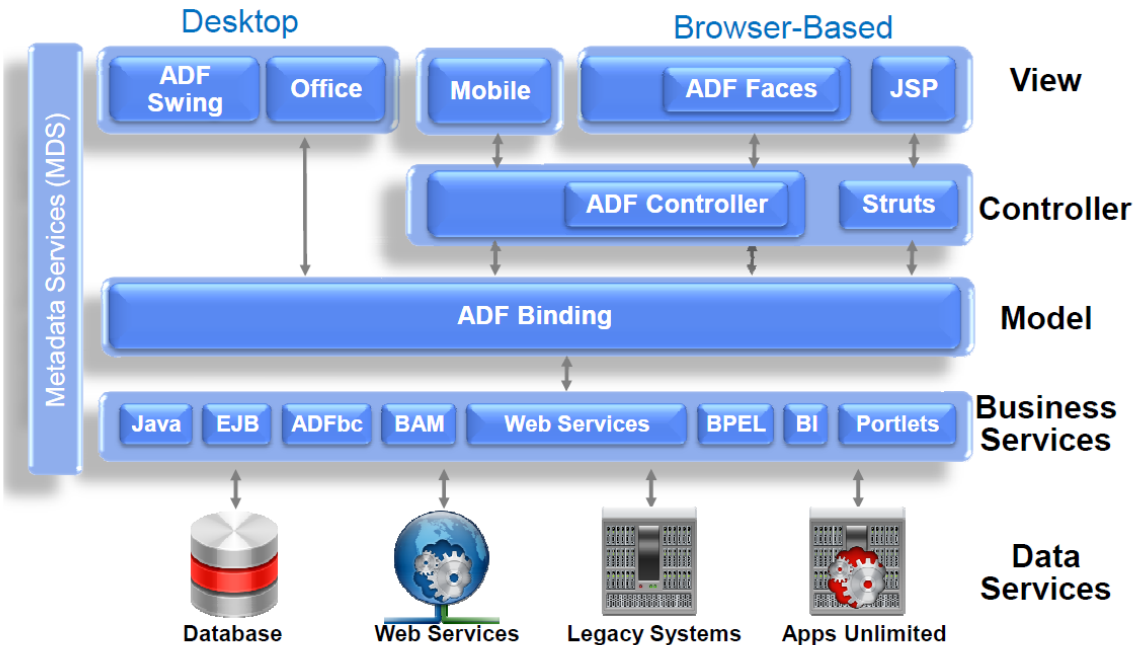
K pochopení co vlastně ADF je, co nabízí a jak funguje, je vhodné popsat si ADF architekturu a jak je rozdělena, jak je rozvrstvena a co jednotlivé části nabízí. Jak je přehledně zobrazeno na následujícím obrázku, je ADF rozděleno na čtyři vrstvy.

V základu je to *Business Service Layer*, tedy vrstva služeb obsahující aplikační logiku, poskytující služby perzistence dat, objektově-relačního mapování, řízení transakcí, komunikace s externími systémy. Je na vývojáři, zda zde využije POJO (Plain Old Java Object), EJB (Enterprise Java Beans), ADF business komponenty, webové služby či jiné.

O úroveň výše se nachází *Model Layer*, tedy vrstva modelu. Jedná se o abstraktní vrstvu, která sjednocuje přístup ke službám Business vrstvy, ať už přistupujeme k EJB nebo ADF business komponentám.

Další vrstvou, která je používaná v aplikacích s tenkým klientem a mobilních aplikacích, je vrstva *Controller*. Ta nám poskytuje mechanismu řízení webové aplikace. Zde je místo pro využití JSF nebo *Apache Struts*. V některých případech ale můžeme tuto vrstvu vynechat.

Poslední vrstvou je *View Layer*. Právě zde je definována výsledná vnější podoba aplikace, zde definujeme prezentaci dat. Nejčastěji se zde setkáme s JSP stránkami, HTML, Servlety atd. V případě ADF zde můžeme využít také širokou paletu ADF Faces komponent, které jsou připraveny řešit většinu běžných požadavků zákazníků, od nejrůznějších formulářů a datových tabulek, až po dynamické grafy, geografické mapy a další.



Obrázek 11 - Architektura Oracle ADF, zdroj: [9]

4.2.1 ADF životní cyklus

ADF Framework rozšiřuje JSF Framework, takže přebírá i dříve uvedený JSF životní cyklus. Ten dále rozšiřuje přidáním životního cyklu na straně klienta. ADF Framework poskytuje konverzi a validaci na straně klienta, která nevyžaduje průchod uvedeným JSF cyklem. Využívá k tomu JavaScript konvertory a validátory. Díky tomu je snížena režie v podobě komunikace se serverem. Také se jedná o uživatelsky přívětivější způsob, protože uživatel nemusí dlouho čekat na odezvu serveru a práce s aplikací je tak plynulejší, pohodlnější a rychlejší. Tento způsob je nyní častý, je využíván například v ASP .NET aplikacích.

S tím také souvisí další funkcionalita implementovaná v ADF a tou je parciální rendering stránek. Jedná se o schopnost komponent nebo oblastí na stránce občerstvit (provést refresh) bez opětovného načítání celé stránky. Komponenty využívají Ajax požadavky (*XMLHttpRequest*). Tuto vlastnost implementuje *ADF renderer* jednotlivých komponent.

Další vlastností je takzvané *Partial Page Refresh*, neboli částečná obnova stránek. V principu se jedná o schopnost vyvolat renderování části stránky z komponenty. Provádí se v rámci ADF Faces životního cyklu a pracuje s renderovanými komponentami. Komponenty na straně klienta musí existovat.

4.2.2 Business komponenty

Business komponenty (ADF Business Components) jsou částí na úrovni Business Service vrstvy, tedy vrstvy aplikační logiky, viz Obrázek 11 - Architektura Oracle ADF. Umožňují na základě informací z relačních databází deklarativním způsobem vytvořit kompletní business vrstvu.

ADF nám pomůže vytvořit Java metody pro obvyklé události, jako je vytvoření, smazání a update záznamů. Na základě relací v databázi vytvoří vazby a omezení na úrovni této vrstvy.

Rámec business komponent zahrnuje dvě skupiny komponent:

- Business doménové komponenty
  - Vynucuje dodržování business pravidel
  - Entitní objekty, asociace entit a vlastní objektové typy
- Data Model komponenty
  - Poskytuje datový přístup do klientské aplikace
  - Objekty pohledů (View Obects), Pohledy na vazby (View Links) a Aplikační moduly

4.2.2.1. Entity Object

Základem jsou entitní objekty, které nám poskytují přístup k datům. Používáme-li k perzistenci relační databázi, poskytnou nám objektově-relační mapování. Jeden entitní objekt zpravidla odpovídá právě jedné tabulce v relační databázi. Entitní objekty pak přebírají a reprezentují vlastnosti těchto tabulek:

- Atributy
- Integritní omezení
- Defaultní hodnoty

Dá se říci, že zapouzdřují business pravidla a vytváří vrstvu nad relační databází a jejími tabulkami, vazbami atd.

4.2.2.2. View Object

Tvoří jakousi další vrstvu nad Entitními objekty. Zapouzdřují SQL dotazy pro projekci, spojování (Join), filtrování, řazení (Order data) pro interakci s externím klientem

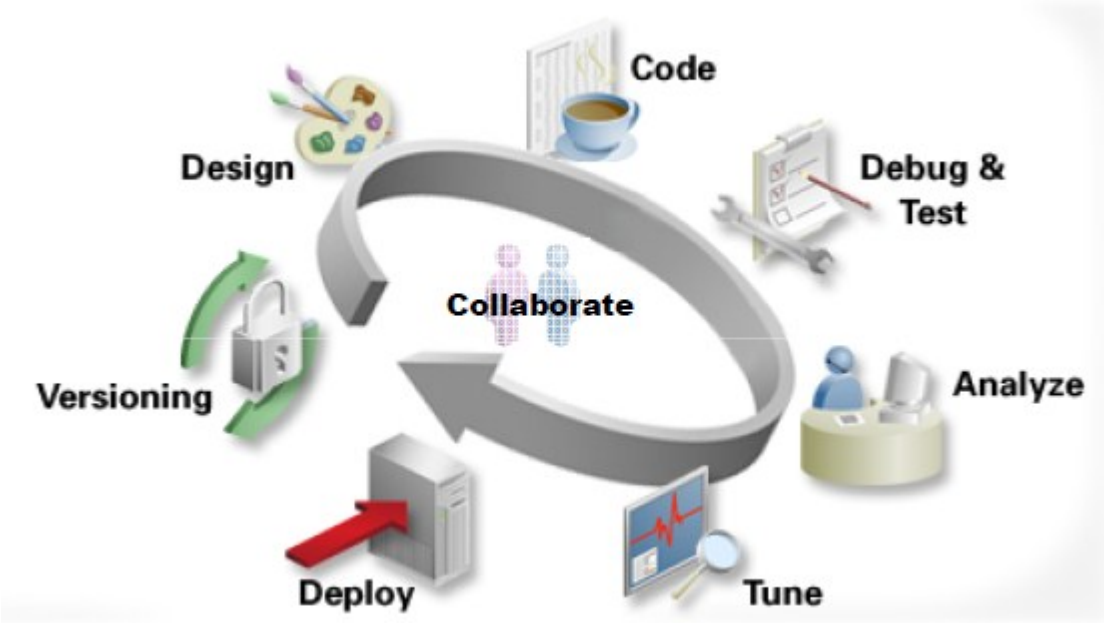
4.2.2.3. Application Module

Podobně můžeme vidět Aplikační modul jako další vrstvu, která se rozprostírá nad View Objekty a využívá ji k tomu, aby definovala datový model použití View objektů pro kompletní případy užití aplikace. Uživatelská rozhraní pracují s aplikačním modulem jakožto jejich back-end business službou.

Komponenty jsou znovupoužitelné, takže je možné vytvořit nový aplikační modul využívající existujících View objektů, třeba jen jiné množiny. Toho se dá využít při tvorbě jiné části jedné aplikace, nebo při tvorbě úplně jiné aplikace. Využití existujících objektů tak ušetří čas a tím i peníze. Pokud byly objekty na začátku dobře navrženy.

4.2.3 Vývoj ADF - JDeveloper

Chceme-li začít s vývojem aplikací s rámcem ADF, je vhodné sáhnout po správném vývojovém prostředí. Nejlepší volbou se jeví *Oracle JDeveloper*, který se řadí do široké škály produktů Oracle Fusion. JDeveloper je volně dostupné integrované vývojové prostředí (IDE) pro vývoj vícevrstvých aplikací. Představou Oracle je nabídnout komplexní vývojové prostředí, které nám nabídne podporu od fáze návrhu, přes návrh grafického rozhraní, vývoj aplikací, až po nasazení hotové aplikace. Samozřejmostí je dnes již podpora testovacích nástrojů a podpora verzí. JDeveloper je také možné použít při tvorbě PL/SQL procedur.



Obrázek 12 - Pokrytí vývoje JDeveloperem

V případě vývoje ADF aplikace nás zajímá podpora tohoto aplikačního rámce. Teprve při práci JDeveloperem pochopíme, kam až došla snaha o intuitivní vizuální a deklarativní vývoj aplikací. Velké množství rutinních činností, které vývojáři zaberou nemalou část vývojového času, se zde dají udělat několika kliknutími myši. Pokud víme jak. V příloze B máme ukázku z aplikace JDeveloper

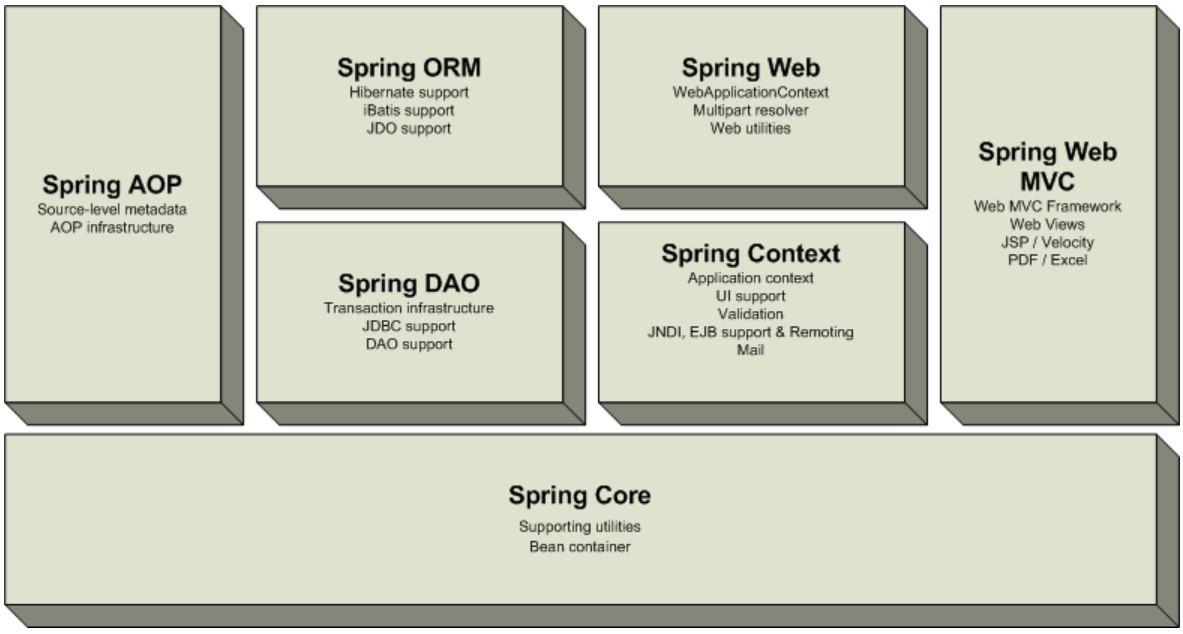
5. Spring framework

Spring framework je velice oblíbený aplikační rámec pro vývoj pokročilých J2EE aplikací. V základech tohoto open-source projektu se nachází kód, který byl publikován v roce 2003 Rodem Johnsonem v [1] a který odráží Johnsonovu několikaletou analytickou, konzultantskou, ale i programátorskou zkušenost s tvorbou rozsáhlých JEE aplikací. Tento kód byl autorem navržen pro zefektivnění řešení některých běžných problémů, se kterými se každý vývojář pokročilých Java aplikací setkává téměř denně, a celkově pro zjednodušení a snížení ceny návrhu a vývoje těchto programů.

Uveďme si, v čem tkví ono usnadnění vývoje:

- Pomoc při odstranění těsných programových vazeb jednotlivých POJO objektů a vrstev za pomoci návrhového vzoru Inversion of Control.
- Máme možnost volby implementace (POJO, EJB) business vrstvy pro aplikační architekturu, místo aby nám ji architektura předepisovala.
- Řešení různých aplikačních domén bez nutnosti použití EJB, například transakční zpracování, podpora pro *remoting* business vrstvy formou webových služeb či RMI.
- Podpora implementace komponent pro přístup k datům, ať již formou přímého JDBC či ORM (object-relation mapping) technologií a nástrojů jako Hibernate, TopLink, iBatis nebo JDO. Mnohými je právě spojení Spring – Hibernate doporučováno jako jedna z nejlepších variant.
- Odstranění závislosti na roztroušených konfiguracích a pracného dohledávání jejich významu.
- Abstrakce vedoucí ke zjednodušenému používání dalších části J2EE, jako například JMS, JMX, JavaMail, JDBC, JCA nebo JNDI.
- Usnadnění používání a psaní unit testů.
- Správa a konfigurační management business komponent.

Další informace naleznete česky v [16] nebo na stránkách projektu www.springframework.org.



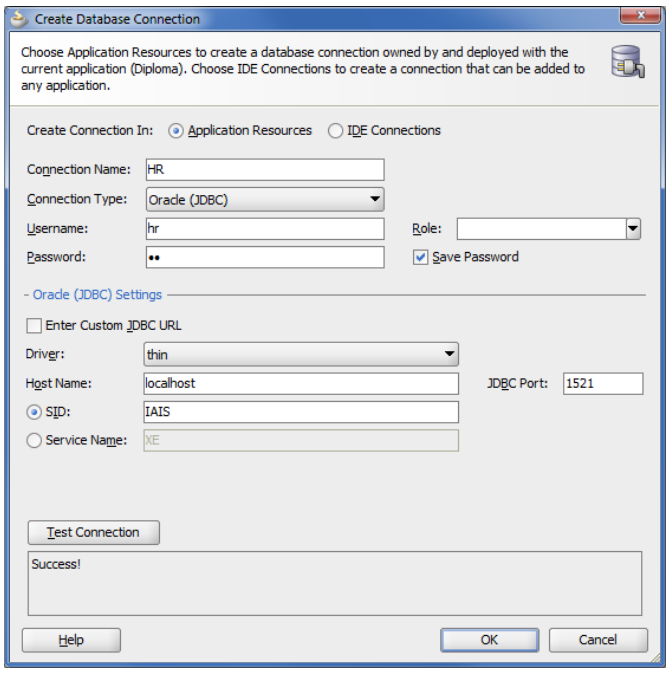
Obrázek 13 - Základní moduly rámce Spring, zdroj: [16]

6. Praktická část

Po představení jednotlivých technologií přejděme k tvorbě ukázkové aplikace, která uvedené technologie vhodně využívá. Jako vývojové prostředí je tedy využito představené IDE Oracle JDeveloper. Díky integraci s WebLogic serverem, který bude poskytovat běhové prostředí aplikaci, je ideální volbou, neboť má mimo jiné plnou podporu pro vývoj za použití představeného aplikačního rámce Oracle ADF. Pro perzistenci dat je nasnadě využít relační databázi Oracle 11g. Ačkoli je použita plná verze databáze, měla by postačovat i odlehčená verze Oracle XE. Důležité je, že obě verze obsahují ukázkové schéma HR, které bylo v této aplikaci využito pouze s malou modifikací v podobě přidaného sloupce pro heslo uživatele.

6.1 Vrstva modelu

Základem je mít vytvořenu vrstvu modelu. Máme-li již navrženu databázi, můžeme na jejím základě model připravit. Musíme mít v aplikaci definováno připojení k databázi, se kterou budeme pracovat. Obrazovka při vytváření spojení s DB je ukázána na následujícím obrázku.



Obrázek 14 - Vytvoření DB připojení

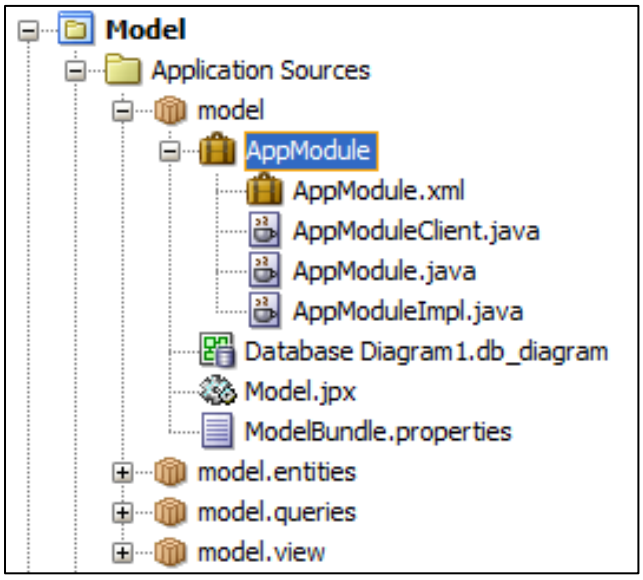
Máme-li definováno připojení k naší databázi, nestojí nám nic v cestě k využití JDevelopera a rámce ADF a vygenerovat v několika krocích základní podobu business komponent a aplikačního modulu. Po načtení informací z databáze si můžeme vybrat, pro které všechny tabulky chceme objekty vytvořit. Ukázku průvodce tvorbou business komponent najdeme v příloze C. Na následujícím obrázku je pak vidět vytvořená struktura, kterou pro nás, na základě naší specifikace, JDeveloper v aplikaci vytvořil. Vytvořený aplikační modul byl pojmenován AppModule.

6.1.1 Aplikační modul

Řekneme-li aplikační modul, znamená to fyzicky několik souborů, v našem případě:

- AppModule.xml
- AppModuleClient.java
- AppModule.java
- AppModuleImpl.java

Obecně bychom si vystačili pouze s konfiguračním souborem AppModule.xml a zbytek bychom mohli nechat na aplikačním rámci, v našem případě to však bylo nedostatečné, viz dále. Struktura aplikace pak vypadá následovně:



Obrázek 15 - AppModule

AppModule.java je pouze rozhraní implementované třídou `AppModuleImpl` (která mimo to rozšiřuje třídu `ApplicationModuleImpl` z balíku `oracle.jbo.server`). Pokud bychom si vystačili s tím, co nám JDeveloper dle našich požadavků vytvořil, budeme pracovat pouze s `AppModule.xml`, kde se nachází konfigurace, které View objekty používáme. Nemusíme samozřejmě pracovat přímo s XML souborem, JDeveloper nám nabízí vizualizaci jeho obsahu rozdělenou do kategorií. Můžeme si zde pod záložkou Data model definovat, se kterými View objekty budeme pracovat.

V naší aplikaci si však s tímto základním modelem nevystačíme, jelikož se chceme do části aplikace přihlašovat uživatelským jménem a heslem. Bylo by možné se k tomuto vrátit až v části o zabezpečení, ale jelikož se jedná o zásah do aplikačního modulu, bude to popsáno již zde. V konfiguraci aplikačního modulu v záložce *Java* se nachází část *Client Interface*. Zde je možné definovat vlastní metody, které pak v prezentační vrstvě využijeme. V našem případě se jedná o metodu `checkUser`. Vytvoříme-li zde novou metodu, je vytvořena na několika místech. Předně zde v konfiguraci (tedy konfiguračním XML), kde se registruje v části client interface:

```
<ClientInterface>
  <Method Name="checkUser"
    MethodName="checkUser">
```

```
            <Return Type="int"/>
            <Parameter
              Name="userid"
              Type="java.lang.String"/>
            <Parameter
              Name="pass"
              Type="java.lang.String"/>
          </Method>
        </ClientInterface>
```

Dále pak v implementaci aplikačního modulu AppModuleImpl.java. A právě zde je tělo metody a zde definujeme její logiku:

```
public int checkUser(String userid, String pass) {
      SAppusersViewImpl myView =
                  (SAppusersViewImpl)getSAppusersView();
     try {
           myView.setNamedWhereClauseParam("employeeId",
                                           new Number(userid));
           myView.setNamedWhereClauseParam("password", pass);
     } catch (SQLException e) {
           e.printStackTrace();
     }
     myView.executeQuery();
     if (myView.getRowCount() == 0) {
        return 0;
     } else {
        return 1;
     }
}
```

Jak tedy vidíme, ani zde se programování úplně nevyhneme, pokud netvoříme pouze triviální aplikaci. Mimochodem tato třída implementuje také *get* metody pro přístup k použitým View objektům. Ty nám ale JDeveloper s ADF již připraví. Například metoda pro načtení View objektu EmployeesManagerView vypadá následovně:

```
public ViewObjectImpl getEmployeesManagerView() {
      return
(ViewObjectImpl)findViewObject("EmployeesManagerView");
}
```

Máme tedy aplikační modul, ale je možné a v praxi obvyklé, že nám vytvořená podoba vygenerovaných objektů zcela nevyhovuje. Ne jinak je tomu u naší aplikace. Nezbývá tedy, než je upravit dle svých potřeb.

6.1.2 Entity objekty

Entitní objekty jsou definovány XML souborem. Ten byl původně vytvořen na základě informací z databáze a z ní také převzal informace o atributech, datovém typu na straně databáze i typu, na který má být převeden na straně aplikace. Jedná se vlastně o informace objektově relačního mapování. XML pak vypadá následovně:

```
<Entity
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="Countries"
  Version="11.1.2.60.81"
  DBObjectType="table"
  DBObjectName="COUNTRIES"
  AliasName="Countries"
  BindingStyle="OracleName"
  UseGlueCode="false">
  <Attribute
      Name="CountryId"
      IsNotNull="true"
      Precision="2"
      ColumnName="COUNTRY_ID"
      SQLType="CHAR"
      Type="java.lang.String"
      ColumnType="CHAR"
      TableName="COUNTRIES"
      PrimaryKey="true"
      ...
  </Attribute>
```

Jak je zde zvykem, nepracujeme přímo s XML (ale můžeme), využíváme grafické prostředí, které je intuitivní a přehledné. Můžeme zde dle své potřeby upravit datové typy, které mají být v aplikaci použity, dále zde odstraníme ty atributy, o kterých víme, že s nimi v aplikaci nebudeme nikdy pracovat.

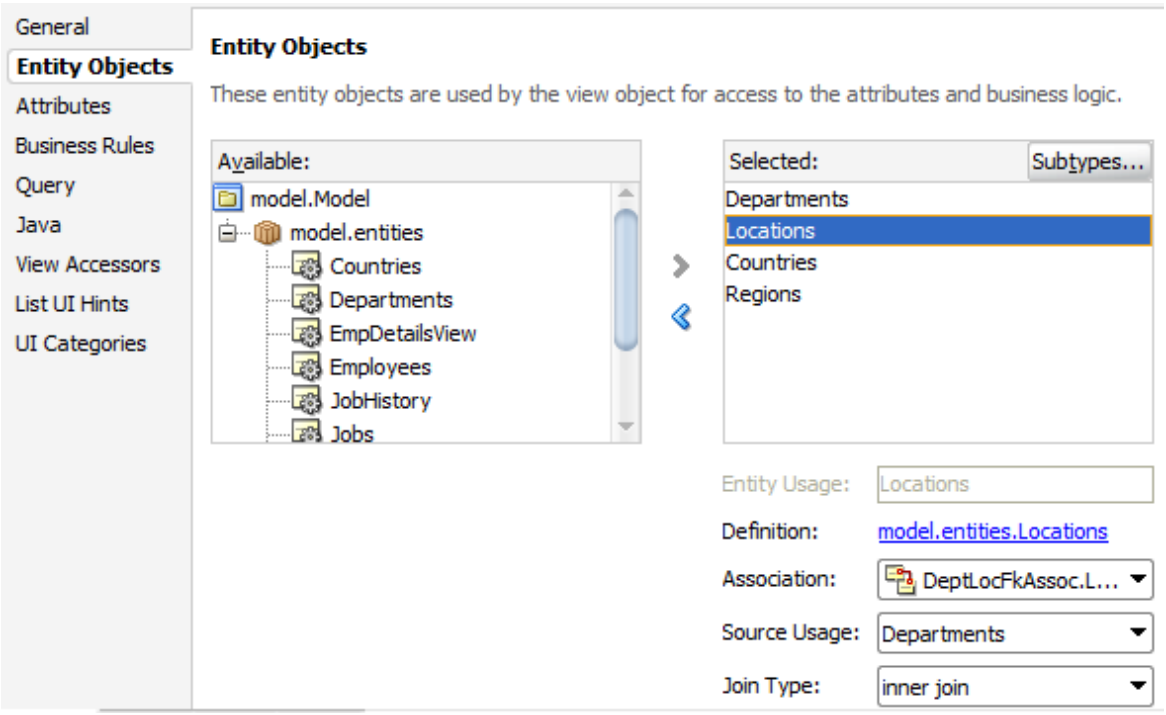
Podobně jako v aplikačním modulu, také zde bylo potřeba upravit si chování ke svému. Mimo základní XML reprezentace můžeme nechat vytvořit implementaci entitního objektu. V našem případě se jedná o objekt *Employees*, kde vedle sebe máme `Employees.xml`, zmíněnou konfiguraci, dále pak EmployeesImpl.java, kde se nachází implementace této entity. Na základě XML je tato implementace vytvořena s generovaným obsahem, jako jsou *get* a *set* metody pro jednotlivé atributy, metody pro vytvoření a rušení záznamu v databázi. Co vše má být generováno a co se ponechá na aplikačním rámci je čistě na nás. V našem případě jsme si nechali vytvořit metodu pro vytvoření záznamu a tu následně upravili dle našich potřeb, aby pracovala se sekvencí.

Entitní objekty jsou tedy opět třídami, tentokráte rozšiřujícími třídu `EntityImpl` z balíku `oracle.jbo.server`.

6.1.3 View objekty

Podobně je to u View objektů. Ty jsou také vytvořeny na základě databázových tabulek a jejich atributů, což ale většinou nestačí. Proto je potřeba si View objekty upravit dle našich požadavků, nebo si vytvořit View objekt úplně nový.

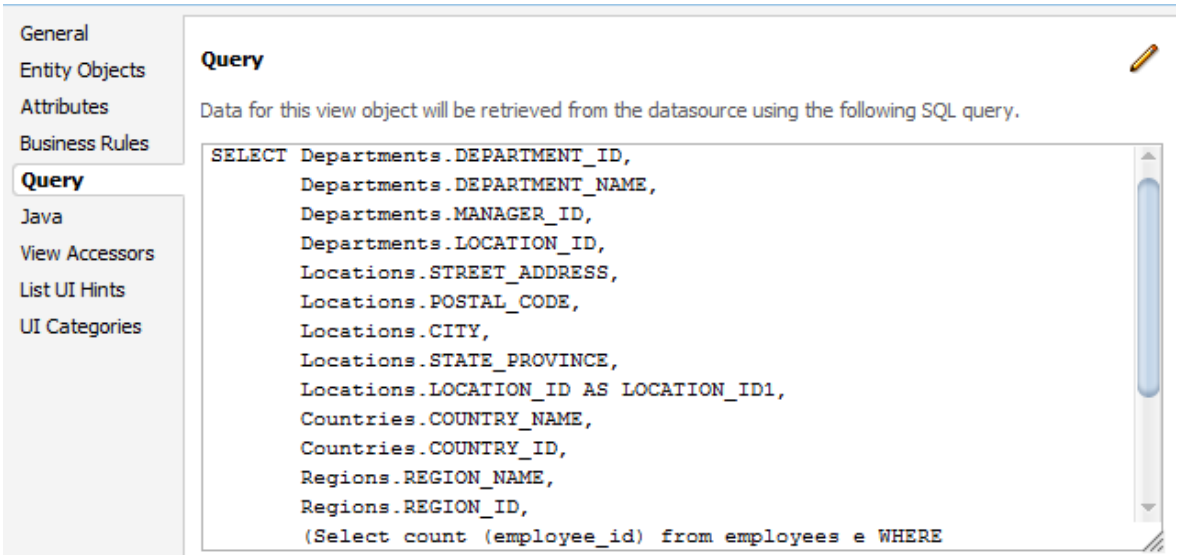
V našem případě byl například View objekt DepartmentsView upraven tak, aby rovnou obsahoval všechny atributy z navázaných tabulek Locations, Countries a Region. To provedeme opět v konfiguraci View objektu. Výsledek pak vypadá následovně:



Obrázek 16 - View objekt Departments - obsažené entity

Jak můžeme vidět, ve View objektu můžeme pracovat s kterýmkoli entitním objektem v rámci daného modelu. Přidáním entit, které jsou v nějaké relaci, definujeme také typ a způsob spojení na úrovni těchto entit. Na základě této deklarace nám je upraven dotaz, viz záložka *Query* na dalším obrázku. Můžeme opět nechat na rámci, aby se o SQL dotaz postaral sám, nebo do něj máme možnost zasáhnout. To je ale závislé na našich potřebách. V případě této aplikace také nebyl původní dotaz ponechán, ale byl upraven tak, aby poskytnul další informace, například počet zaměstnanců v dané pobočce, čehož je využito později.

Chceme-li rozšířit množinu atributů View objektu, můžeme tak udělat na záložce Attributes. Dotaz nám poskytne data, ale ta chceme nějak prezentovat a právě zde je místo, kde definujeme, jak se atribut bude jmenovat, jakého typu bude a kde pro něj získat data, zda z SQL dotazu, jak je tomu zde, nebo například na základě výrazu, který data vytvoří například operací nad některým z ostatních atributů.



Obrázek 17 - SQL dotaz pro načtení dat do View objektu

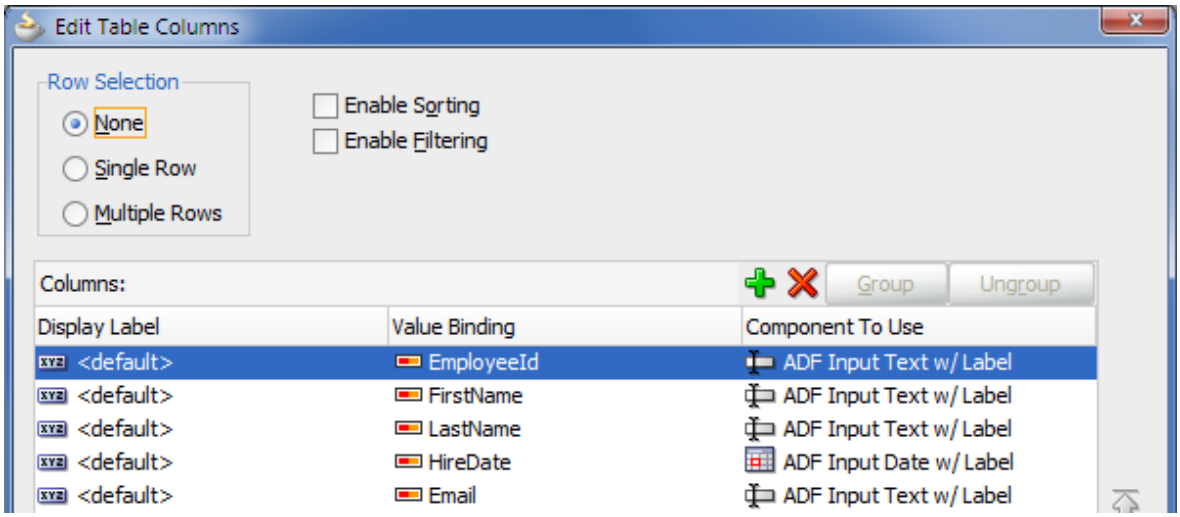
Také zde můžeme aplikačnímu rámci přenechat práci s tvorbou vlastní implementace, nebo můžeme požádat o vygenerování implementační třídy daného View objektu a poté doimplementovat, co je třeba. Nutno podotknout, že se tomu mnohdy nevyhneme.

V případě naší aplikace bylo mimo jiné vytvořeno zcela nové View, které nám poskytuje informace pro přihlašování uživatele. Pojmenováno bylo SAppusersView a pracujeme zde pouze s údaji týkajícími se s přihlašováním uživatelů, s ničím jiným.

Velkou výhodou použitých vývojových nástrojů je možnost definované objekty hned otestovat. Není potřeba vytvářet testovací stránky, kde bychom prováděli testování, nebo nechávat odzkoušení až na okamžiku použití objektu. Chceme-li, stačí vybrat aplikační modul a zvolit z kontextové nabídky Run. JDeveloper nám zobrazí objekty zvolené v aplikačním modulu v podobě formulářů a tabulek, master/detail pohledy, pokud byly připraveny.

6.2 Vrstva ViewController

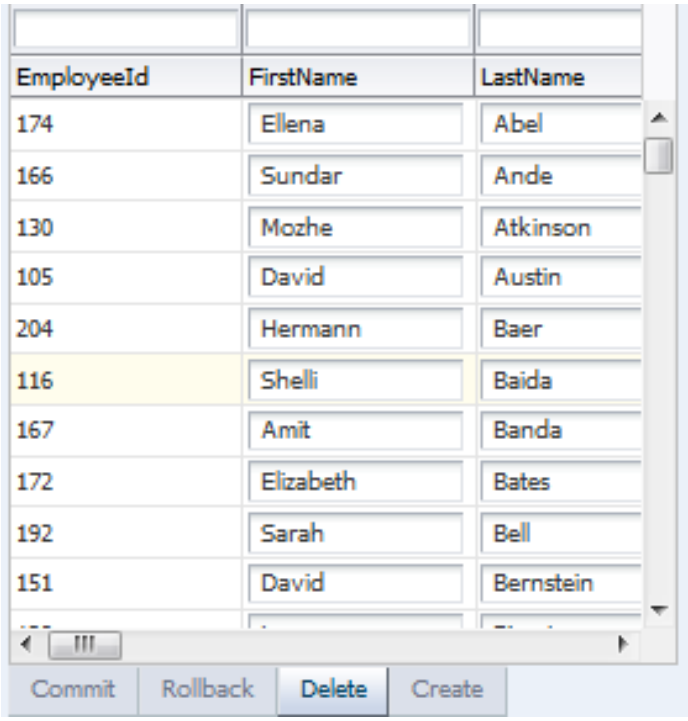
Máme-li připraven model, můžeme přejít k vytvoření prezentační vrstvy. Oracle ADF nabízí přes 150 JavaServer Faces komponent s podporou AJAX. Jednoduše pak můžeme vytvořit například tabulku záznamů, kterou bychom pomocí jiných technologií připravovali přeci jenom delší chvíli. Tady ale pouze využijeme vytvořených View objektů, které byly vytvořeny v předešlém kroku. Stačí otevřít paletu *Data Controls*, kde jsou připraveny. Stačí pak pouze přetáhnout například EmployeesView do předem připravené jsp nebo jspx stránky a zvolit komponentu, kterou chceme nechat vygenerovat. V našem případě jsme si vytvořili stránku employees.jspx a do ní vložili komponentu *ADF Table*. Jak je vidět na dalším obrázku, během vytváření máme možnost upravit výsledný vzhled datové tabulky a také přidat možnosti řazení a filtrování, které jsou často zákazníkem požadovány. Je možné dále setřídit atributy dle požadavků, změnit typ generované komponenty pro jednotlivé atributy (například zda půjde o textový input, output text, nebo třeba speciální vstup pro datumovou položku). Lze také definovat labely pro jednotlivé elementy. To vše je samozřejmě možné upravit i po vygenerování, neboť ADF toto vše vloží do stránky.



Obrázek 18 - přizpůsobení vytvářené datové tabulky

V závislosti na zvolených atributech pak tabulka vypadá následovně. Obrázek zobrazuje tabulku, která obsahuje také pole pro filtrování a možnost řazení. V tomto případě byla zvolena tabulka s textovými vstupy, takže uživatel může během prohlížení data editovat. Změny se promítnou až po jejich odeslání do databáze – commit. I toto je v aplikaci využito. V paletě Data Controls jsou operace, které mohou být přidány na stránku – Commit a Rollback. Jejich význam je zřejmý. Vložením těchto operací pod tabulku, ať už jako link nebo tlačítko, můžeme nechat v tabulce provedené změny promítnout do databáze, nebo je zrušit a načíst původní hodnoty.

Podobné operace jsou také uvnitř EmployeesView. V našem případě byla použita operace delete, takže po zvolení libovolného řádku v tabulce můžeme tento nechat odstranit. Podmínkou je, že je výběr řádku povolen (například single row sellection).



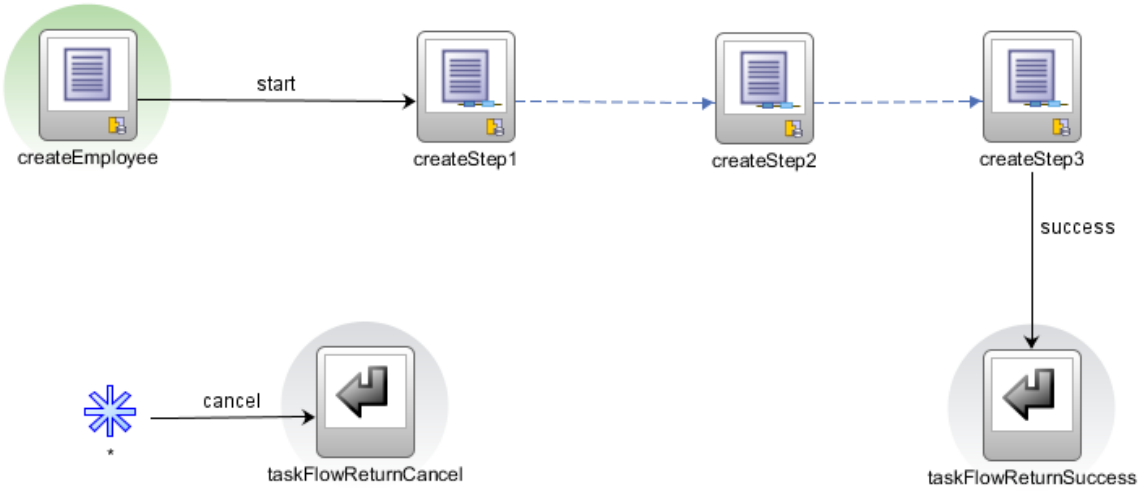
Obrázek 19 - Vytvořená datová tabulka

Vytvoříme-li na stránce nějakou komponentu, není jspx stránka jediným místem, kde se to projeví. Ke stránkám náleží definiční soubory XML, například ke stránce employees.jspx je to definice view/pageDefs/employeesPageDef.xml. Zde je obsažena definice použitých atributů a jejich binding na datové kontrolky a další.

6.2.1 Task Flow

Mnohdy potřebujeme definovat předem daný sled úkonů, které mají být provedeny. K tomu zde máme k dispozici Task Flow. Jedná se o znovu použitelný, parametrizovatelný způsob navigace v aplikaci mezi zobrazovacími i nezobrazovacími akcemi. Je možné k němu přistupovat jak lokálně, tak vzdáleně.

V aplikaci je task flow použito pro definování procesu vytvoření nového uživatele. Postupně přecházíme mezi kroky createEmployee, createStep1, createStep2 a createStep3, přičemž můžeme kdykoli proces zrušit pomocí zavolání cancel. Stav taskFlowReturnSuccess znamená potvrzení operace a úspěšný návrat do místa, odkud byl proces zavolán.



Obrázek 20 - TaskFlow - Vytvoření záznamu zaměstnance

6.2.2 Realizace přihlášení

Málokterá aplikace se obejde bez správy uživatelů. Ukažme si, jak je zde řešeno přihlašování. Jak již bylo uvedeno dříve, budeme využívat data v tabulce Employees, sloupce EMPLOYEE_ID a námi přidaný sloupec PASSWORD. Jak bylo uvedeno v části o aplikačním modulu, bylo potřeba tento rozšířit o metodu, která nám ověří přihlašovací údaje. Metoda se jmenuje `checkUser`. Máme-li toto k dispozici, můžeme si vytvořit přihlašovací stránku. Nejedná se o jediné řešení security, ale o tom dále. Vytvořili jsme si proto stránku login.jspx. Z nabídky Data Controls si přetáhneme metodu `checkUser` jako ADF Parameter Form. Tím získáme formulář se vstupy pro uživatelské jméno a heslo, dále pak tlačítko pro odeslání. Po úpravě jednotlivých elementů je potřeba propojit tlačítko pro přihlášení s logikou. Postačí dvojklik na tlačítko a zobrazí se nám možnost, kde necháme vytvořit metodu cmdLogin_action v managed bean ‚backing_login' ve třídě Login.java. Její obsah je zobrazen v příloze D.

6.2.3 Zabezpečení aplikace

K zabezpečení aplikace je možné použít Java Authentication and Authorization Service (JAAS). Bezpečnostní logika je oddělena od logiky aplikační. Zabezpečit celou aplikaci je díky použitým nástrojům opět poměrně jednoduché, pokud víme jak na to.

Nejjednodušší cestou, která byla zvolena i v naší aplikaci, je vybrat z hlavní nabídky Application > Secure > Configure ADF Security, čímž spustíme *Configure ADF Security wizard*. Vybereme ADF Authentication and Authorization. V části Select authentication type si můžeme vybrat mezi způsoby autentikace. Často využitelnou možností je Form-Based Authentication, kde si můžeme přihlašovací a chybovou stránku vybrat z existujících, nebo můžeme nechat JDevelopera, aby nám je vytvořil. Dále můžeme definovat, zda chceme aplikaci kompletně zabezpečit, což je vhodné, nebo zda si chceme dodatečně definovat části, které budou zabezpečené. V našem případě bylo použito možnosti, aby se na stránky dostal pouze přihlášený uživatel.

Je možné definovat také uživatelské role a jednotlivé části zpřístupnit jen některým uživatelským skupinám. Toto a mnoho dalšího je možné najít na stránkách rámce ADF.

6.2.4 Spring

Nejprve je potřeba zmínit, že JDeveloper má podporu pro aplikační rámec Spring a je tedy možné využít jeho služeb při vývoji aplikací. Během vývoje ukázkové aplikace při použití Oracle ADF však některé místa, která jsou běžně řešena za použití tohoto rámce, byla řešena rámcem ADF pro jednoduchost použití. Příkladem může být například využití ADF pro mapování na databázi, místo použití JDBC Template či jiných způsobů mapování. Jsou tedy stále místa, kde můžeme Spring použít, ale je otázkou, zda je jeho použití lepší než využití aplikačního rámce ADF. Z toho důvodu nakonec Spring využit v ukázkové aplikaci není.

6.2.5 Problémy

Ačkoli se jedná o skvělou technologii, ani ta není bez chyby. Během přípravy aplikace bylo například nemožné vytvořit konfiguraci pro mapy, které jsou jednou z komponent. Řešením byl update verze JDevelopera. Další možností bylo vytvořit tuto konfiguraci ručně, ale tím, jak nám ADF nabízí programování bez nutnosti znát vše, co se děje na pozadí, je to práce spíše pro zkušenějšího ADF vývojáře.

Během vývoje se vyskytoval problém s nedostatkem paměti. Stačilo pustit několikrát re-deployment a server upozorňoval na hrozící nebezpečí OutOfMemory v oblasti haldy. Příčiny a řešení těchto problémů je však problematika na celou další kapitolu.

7. Řešení problému s nedostatkem paměti na platformě Java

Nedostatek dostupné paměti způsobující vyvolání chyby OutOfMemory je problém, který se čas od času objeví na některých aplikacích. Zvláště rozsáhlé enterprise aplikace vytváří široký prostor pro výskyt tohoto problému. O co se tedy jedná? Obecně je to stav, kdy dojde k vyčerpání veškeré dostupné paměti, která byla procesu přidělena. V dřívějších dobách bylo nutné, aby si programátor hlídal odstraňování nepotřebných datových objektů. Jako příklad si zde můžeme uvézt ukazatele odkazující na již nepoužívané objekty, nebo již zrušené objekty. Pokud si toto programátor důsledně nepohlídal, zůstávaly mu tyto v paměti a dále zabíraly místo potřebné k běhu programu. Na platformě Java již toto není nutné, protože Java (ve spolupráci s JRE) spravuje paměť automaticky. Tento proces označujeme jako *garbage collecting*, tedy sběr smetí, kde smetím chápeme právě již nepotřebné objekty.

7.1 Garbage collecting

Pojem *Garbage collecting* (dále GC) poprvé použil roku 1959 John McCarthy, když řešil problém manuální správy paměti v prostředí jazyka *Lisp* (jehož je tvůrcem). Jedná se o snahu automatizovat správu paměti běžícího programu. Zjednodušeně řečeno proces GC hledá nepoužívané datové objekty, které již nebudou potřeba a mohou být z paměti odstraněny, dále pak vrací zdroje, které byly těmito objekty alokovány. Je to rozdíl oproti jazykům jako C, kde si programátor musí sám po sobě uklidit, co už nebude potřebovat. Místa v paměti zabíraná proměnnými, které již program nepoužije, nazýváme *memory leaks*.

Kdyby nebyly používány mechanismy automatické správy paměti jako *garbage collecting*, byl tento problém mnohem častější. Avšak ani s využitím těchto mechanismů se těmto situacím zcela nevyhneme, zvláště pak při vývoji a provozování rozsáhlejších aplikací. Příčin tohoto problému může být více, výsledek je ale totožný – proces zahlásí chybu a nemůže dále pokračovat. V těchto chvílích si již nevystačíme s informací, že Java (ve spolupráci s JRE) spravuje paměť automaticky, ale musíme se touto problematikou zabývat podrobněji.

Jak již bylo uvedeno, základní algoritmus GC lze shrnout do následujících kroků:

1. Vyhledání takových datových objektů v programu, které již nebudou použity.
2. Vrácení zdrojů, které nalezené objekty zadržovaly.

Jakým způsobem je toto realizováno? S postupem času vzniklo několik algoritmů řešících uvedené. Jejich postupným popisem a uvedením výhod a nevýhod pak pochopíme, proč a jak Java postupně změnila práci s pamětí.

7.1.1 Algoritmus počítání referencí

První algoritmus realizující GC pracoval na principu počítání referencí na objekt.

Popis algoritmu počítání referencí:

- Každému objektu je přiřazen čítač referencí. Ihned po vytvoření objektu je jeho čítač nastaven na hodnotu 1.
- Pokaždé, když si nějaký jiný objekt vytvoří referenci na tento objekt, hodnota čítače je zvětšena o 1.
- Je-li reference mimo rozsah platnosti (např. po opuštění metody, která si referenci uložila), nebo je-li referenci přiřazena nová hodnota, čítač je snížen o 1.
- Jakmile dosáhne čítač některého objektu nulové hodnoty, může být tento objekt odstraněn z paměti.
- Při uvolňování objektu z paměti se projdou všechny objekty, na které má tento objekt reference. Všem těmto objektům je pak snížena hodnota čítače o 1. Tím může dojít k uvolnění dalších (původně odkazovaných) objektů a tedy místa v paměti.

Nevýhody:

- Zvýšená režie kvůli nutnosti zvyšovat a snižovat čítač objektu.
- Neschopnost detekce cyklických závislostí (*deadlock*).

7.1.2 Trasovací algoritmy

Tento typ algoritmů, anglicky známých jako *tracing algorithms*, většinou zastaví běh programu a začne od kořene referencí procházet odkazované objekty, které postupně označí. Jakmile projde celý strom, všechny neoznačené objekty jsou chápány jako nedosažitelné a tudíž nepotřebné. Tento způsob byl poprvé využit v jazyce Lisp (roku 1960) algoritmem zvaným *Mark & Sweep*.

Popis algoritmu Mark & Sweep:

- Všem objektům v paměti je nastaven příznak *visited* na *false*.
- Projdou se všechny dosažitelné objekty.
- Dosažitelným objektům je postupně nastavován příznak *visited* na *true*.
- Jakmile se projde celý strom, objekty s příznakem *visited = false* jsou odstraněny.

Nevýhody:

- Přerušení běhu programu.

Výhody:

- Nevadí cyklické závislosti.

Na podobném principu pracuje také kopírovací algoritmus, který však počítá s rozdělením haldy (heap, viz dále) na 2 totožné části, aktivní a neaktivní. Program pracuje pouze s aktivní částí. Která to je, určuje GC, tedy zmíněný kopírovací algoritmus.

Popis ‚Kopírovacího algoritmu' :

- Je zastaven běh programu.
- Postupně se v aktivní části od kořene prochází všechny dosažitelné objekty a jsou kopírovány do části neaktivní.
- Po průchodu celým stromem je aktivní část paměti označena jako neaktivní a naopak.
- Je obnoven běh programu a aplikace pracuje opět s aktuálně aktivní částí paměti, ve které jsou po doběhnutí GC pouze aktivní objekty.

Nevýhody:

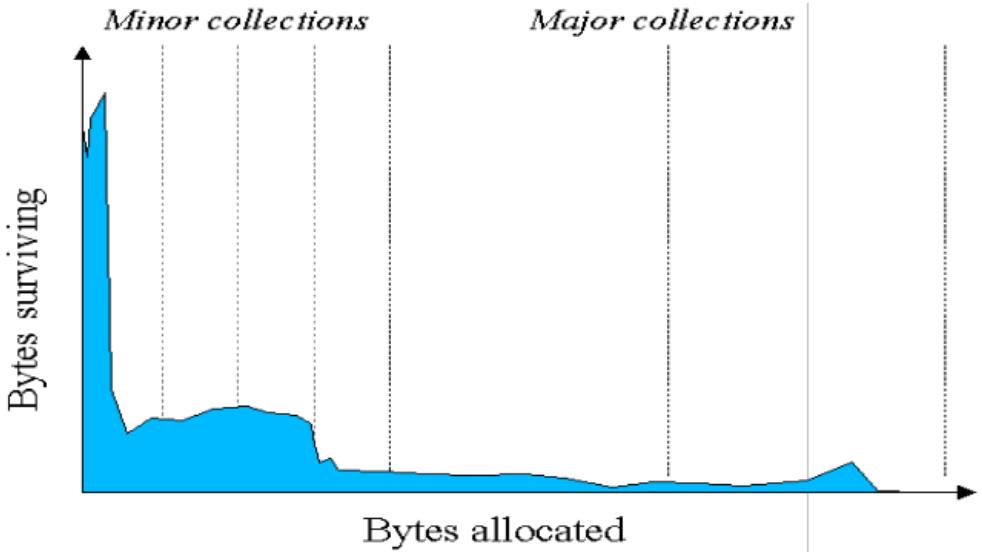
- Přerušení běhu programu.
- Zdlouhavé kopírování objektů.
- Můžeme alokovat objekty v celkové velikosti pouhé poloviny velikosti haldy.

Výhody:

- Nevadí cyklické závislosti.
- Nedochází k fragmentaci.
- K adresaci postačí mnohem menší rozptyl adres.

7.1.3 Generační algoritmus

Postupem času bylo zjištěno několik zajímavých a v této souvislosti důležitých faktů. Prvním je, že mnoho objektů zanikne krátce po svém vzniku. To není ani tak překvapivé, když si uvědomíme, kolik objektů potřebujeme jen pro několik řádků kódu a jakmile svůj účel splní, dále je nepotřebujeme. Dalším zjištěním bylo, že jen minimum starších objektů, tedy objektů přežívajících v paměti po několika běžích GC, odkazuje na objekty mladé.



Obrázek 21 - Množství alokované paměti v závislosti na počtu GC, zdroj: [7]

Na základě těchto skutečností postupně vznikl generační algoritmus. Ten, jak již název napovídá, rozděluje paměť na několik částí, tzv. generací a objekty pak umisťuje do jednotlivých částí právě podle toho, kolik běhů GC v paměti již přečkaly, tedy podle jejich stáří. Díky tomu pak

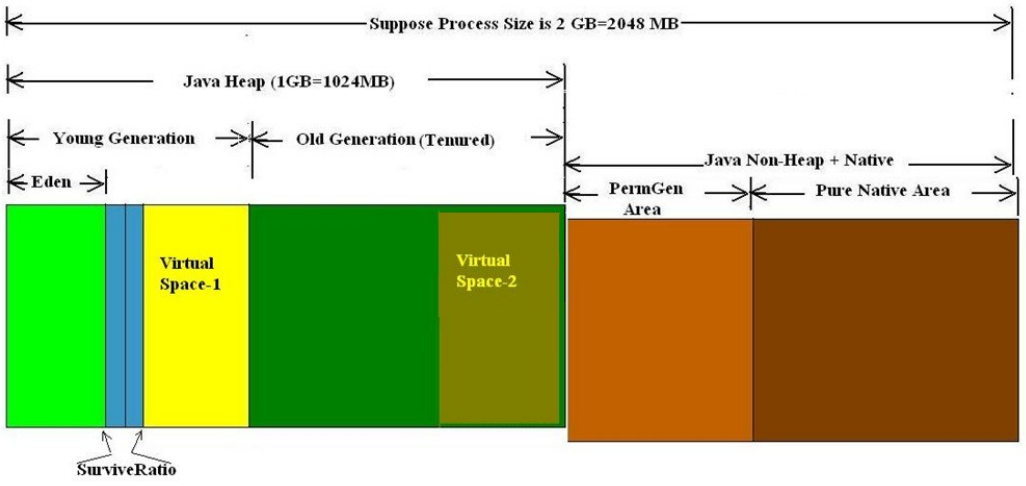
můžeme nad jednotlivými generacemi pouštět GC různě často, což výrazně zvýší efektivitu správy paměti. Stáří objektu nemusí být jediným atributem, podle něhož jsou objekty rozdělovány. O tom ale později.

Existuje mnoho implementací tohoto algoritmu. Jen v rámci Javy 5 se nachází 4 implementace.

7.2 Správa paměti u HotSpot Java VM

Jak již bylo nastíněno v předchozí kapitole, Java využívá k automatické správě paměti generační garbage collector. Podle toho je tedy paměť přidělená Java procesu rozdělena. Při spuštění procesu dojde také k přidělení paměti. Java nám umožňuje velikost přidělené paměti, ale i jednotlivých částí, nastavovat pomocí parametrů (viz dále). Pokud se parametricky neovlivní, spustí se proces s defaultním nastavením v závislosti na konkrétní implementaci JRE.

Popišme si nyní jednotlivé části paměti na následujícím obrázku (Obrázek 22 - Části paměti přidělené Java procesu, zdroj: [6]). Mezní pro jakýkoli proces je celková velikost jemu přidělené paměti. Ta se v základu dělí na 2 části: *Java Heap* (neboli halda) a *non-heap* část. Nutno dodat, že *non-heap* část není jednou jedinou oblastí, ale zahrnujeme zde několik dalších oblastí, viz dále.



Obrázek 22 - Části paměti přidělené Java procesu, zdroj: [6]

7.2.1 Heap

V běžných aplikacích je většinou *Heap* (česky halda) tou částí, o kterou se začneme zajímat, je-li paměť přidělená procesu příliš malá pro jeho běh. Právě zde vznikají nové objekty, zde s nimi pracujeme. Můžeme nastavit její inicializační velikost, tedy velikost při startu procesu. Dále můžeme nastavit její maximální velikost.

Parametry:

- -Xms1024m inicializační velikost haldy v bytech, musí být násobek 1024 (zde 1024MB)
- -Xmx1024m maximální velikost haldy (zde 1024MB)

Zde jsou záměrně použity stejné hodnoty těchto parametrů. Je to poměrně běžnou praxí například na aplikačních serverech v produkčním prostředí, kdy hned po startu procesu alokujeme maximum paměti, aby později při běhu nebyl ovlivněn výkon serveru jejím rozšiřováním. Velikost je závislá na konkrétní aplikaci a v případě aplikačních serverů také na aplikacích na něm nasazeným (*deployed*). Musíme mít na paměti, že aplikační server sám o sobě (s defaultními službami) potřebuje ke svému běhu dostatek paměti. Například Oracle WebLogic Server 11g si vezme 256MB. S dalšími aplikacemi pak může být vhodná velikost haldy v řádech GB.

Jak již bylo zmíněno v popisu generačního algoritmu GC, halda je rozdělena na 2 části dle stáří objektů:

- *Young Generation* pro nově vzniklé a mladé objekty,
- *Old Generation* pro starší objekty, označovaná také jako *Tenured space*.

7.2.1.1. Young Generation

Young Generation je tedy místem, kam se umisťují nově vzniklé objekty. Napíšeme-li v kódu:

```
String s = "text";
```

vznikne instance třídy String právě zde. Je to místo, kde většina objektů rychle zanikne a jsou odstraněny pomocí Minor garbage collectoru. Někteří přeživší zůstávají po určitou dobu, dokud nejsou přesunuty do *Old Generation space*, nebo nezaniknou. O tom, jak dlouho se zde nachází, rozhoduje konkrétní implementace GC a jeho nastavení. Obecně ale můžeme říci, že je *Minor garbage collector* spouštěn poměrně často, takže nepotřebné objekty jsou rychle odstraňovány a paměť přitom není fragmentována. Vlastnosti této části opět můžeme nastavit parametry VM.

Parametry:

- -Xmn100M – velikost Young Gen
  - Ve verzích před JDK1.4 se používá -XX:NewSize a -XX:MaxNewSize
- -XX:NewRatio=<value> - preferováno (dynamické přidělování)

Parametr NewRatio si zaslouží bližší vysvětlení. Uvádí poměr mezi Young generation a Old generation. -XX:NewRatio=6 znamená, že paměť přidělená mezi tenured space a young generation bude v poměru 6:1. Defaultní hodnoty se mezi jednotlivými platformami liší, na což je dobré myslet při nasazování na testovací či produkční prostředí a při řešení problémů na nich, jelikož se mnohdy liší od vývojových.

Ve snaze zefektivnit práci nad *Young generation* byla tato část paměti dále rozdělena na následující:

- *Eden space* – část, kde jsou umístěny nově vzniklé objekty a kde také většina umírá,
- *From space*,
- *To space*.

Jakmile je prostor *Eden* zaplněn, je zavolán Minor GC. Mrtvé objekty jsou odstraněny, přeživší jsou přesunuty do Survivor space. Nad *Survivor* částmi *From space* a *To space* pracuje kopírovací algoritmus, přičemž *To space* je prostor, kam se přesunou živé objekty z oblasti *Eden space* a dále přeživší objekty z *From Space*. Pro další běh *Minor GC* se označení těchto dvou (stejně velkých oblastí) prohodí. Tímto je zajištěno, že nedochází ke fragmentaci přiděleného prostoru.

Zvláště vedle výše uvedených tří částí uvádíme *Virtual Space-1*, což je vlastně rozdíl mezi inicializační hodnotou *Young generation* a její maximální hodnotou v rámci daného prostoru přiděleného procesu. Jinými slovy je to rozdíl mezi -XX:MaxNewSize a -XX:NewSize.

Poznámka:

Objekty, pro jejichž alokaci je zapotřebí velké množství paměti, však mohou být přímo vytvořeny v oblasti *Old generation*.

7.2.1.2. Old Generation

Druhou velkou částí haldy je *Old Generation*, neboli *Tenured space*. Zde jsou postupně přesunuty objekty, které přežily určitý počet běhů *Minor garbage collectoru*. Jakmile potřebuje tato část GC, spouští se takzvaný *Major garbage collector*, který je často mnohem pomalejší, jelikož při své práci zahrne všechny objekty v haldě.

7.2.2 PermGen Space

Další částí paměti, která nás bude zajímat, je Permanent Generation, resp. Permanent Space. V této oblasti jsou uloženy definice tříd, definice metod a další meta data. Zatímco v haldě jsou tedy instance tříd, zde jsou uloženy jejich definice. U běžných Java aplikací jsou zde třídy nahrány pouze 1x. V haldě tedy máme libovolný počet instancí některé třídy, ale její definice je v Permanent Space uložena pouze jednou. JVM proto musí zaručit, že daná třída je nahrána pouze jedenkrát v doménovém prostoru určeném pro *classloader*, což nám zaručí, že se její byte code časem nezmění. Toto je pravidlem, které ale v případě J2EE aplikací nemusí platit. Je to dáno specifikací a tím, jak je v aplikačním serveru uspořádána hierarchie Class Loaderů. Jde o to, že například různé webové aplikace běžící na serveru mohou mít různé verze stejných tříd. Více o tom v části věnované práci Class Loaderů.

Nad touto částí nám defaultně garbage collector neběží, proto se nám právě u Enterprise aplikací stává, že jsou třídy po několika redeploymentech zavedeny opakovaně a prostor přidělený *Permanent generation* je zaplněn. To se projeví chybou java.lang.OutOfMemoryError: PermGen space. V předchozích verzích Sun Javy a i nyní v některých distribucích například od IBM je zobrazena pouze chyba java.lang.OutOfMemoryError bez vysvětlení, ve které části k ní došlo – zda v haldě, nebo v Permanent Space. Obecně se ale s tímto problémem setkáváme velmi často na vývojovém prostředí, kde probíhá redeployment velice často. Zde se to většinou řeší nastavením velikosti této části.

Parametry:

- - XX:PermSize=<value> (inicializační hodnota – po startu procesu)
- -XX:MaxPermSize=<value> (maximální hodnota za běhu procesu)

- -XX:+CMSClassUnloadingEnabled (povolí odstranění nevyužívaných tříd)
- -XX:+CMSPermGenSweepingEnabled

7.2.3 Native Area

Poslední částí paměti přidělené procesu je *Native Area*. Tuto oblast obvykle využívá Java Virual Machine (JVM) pro své interní operace a vykonávání JNI operací. JVM využívá nativní paměť k optimalizacím kódu a k načítání tříd a knihoven spolu s generováním intermediálního kódu.

Velikost této části paměti je přímo závislá na architektuře operačního systému a množství paměti přidělené haldě. Je to procesní paměť, kde se nahrávají JNI kódy, JVM knihovny, nebo také nativní ‚Performance packs' a Proxy moduly.

Nemáme k dispozici JVM parametr pro nastavení velikosti celé této oblasti (některé části ale můžeme ovlivnit parametricky, nebo v aplikačním kódu). Její velikost však můžeme přibližně spočítat následovně:

NativeMemory = (ProcessSize – MaxHeapSize – MaxPermSize)

Do této části zahrnujeme následující prostory:
- Code Generation,
- Socket Buffers,
- Thread Stacks,
- Direct Memory Space,
- JNI Code,
- Garbage Collection,
- JNI Allocated Memory.

7.3 Odhalování problémů s pamětí

Ať už pracujeme na vývoji J2EE aplikací nebo na jejich podpoře, dříve či později narazíme na problém s pamětí. Může se jednat „pouze" o výkonnostní problém, nebo nám čas od času aplikační server spadne na OutOfMemory. V těchto situacích je znalost práce Garbage Collectoru a správy paměti v Javě velice důležitá. Je možné setkat se s řešeními nezakládajícími se na uvedených znalostech, ale výsledky jsou pak spíše náhodné.

Co tedy dělat, pokud nám aplikační server vypíše do logu java.lang.OutOfMemoryError? Pokud používáme Java HotSpot™ VM, dostaneme také informaci, zda k problému došlo v Heapu, nebo PermGen. To je poměrně důležité, protože na platformách, kde se tato informace nevypisuje, jde o první věc, kterou musíme zjistit. Co ale dále? Prvním krokem a nejrychlejším (dočasným) řešením je zvýšit velikost dané oblasti a server restartovat.

Například:
- java.lang.OutOfMemoryError: Java heap space
    - –Xmx512m

- java.lang.OutOfMemoryError: PermGen space
    - –XX:MaxPermSize=256m

Toto skutečně může být řešením, pokud byla velikost dané oblasti nevhodně malá. Ve většině případů to ale problém pouze oddálí. Co tedy s tím? V tu chvíli přichází na řadu diagnostické nástroje. První a nejjednodušší volbou bývá *jmap*. Ve většině případů však ze zjištěných informací příčinu nezjistíme a musí přijít na řadu pokročilejší a sofistikovanější nástroje. Při řešení těchto problémů v praxi na produkčních prostředích se velice osvědčil Eclipse memory analyzer.

7.3.1 Eclipse Memory Analyzer

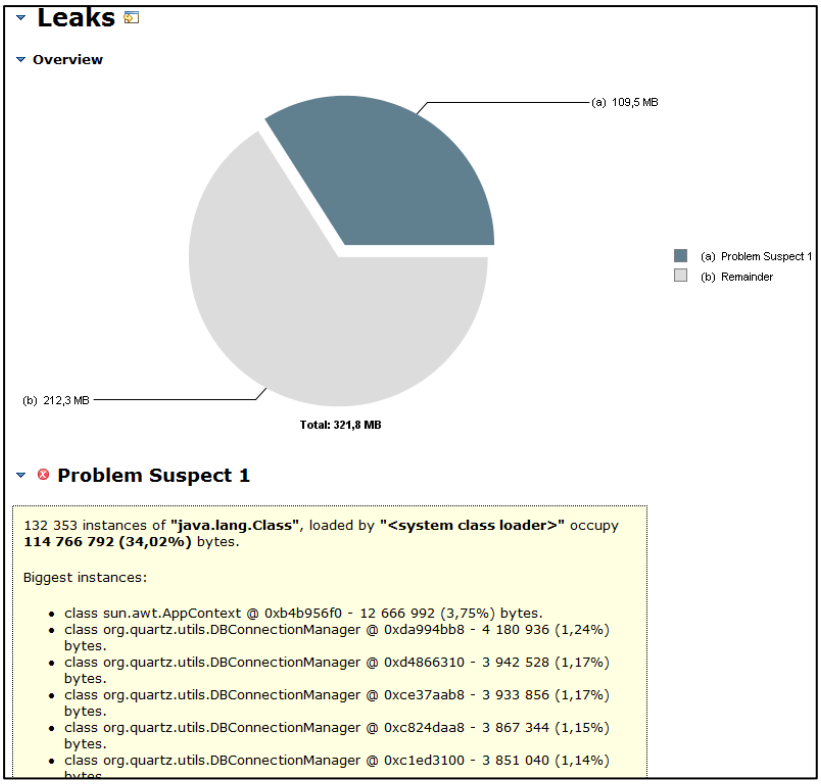
Velice užitečným nástrojem při řešení těchto problémů je *Eclipse memory analyzer* (zkráceně MAT). S jeho vývojem se začalo ve společnosti SAP, proto se můžeme setkat také s označením *SAP Memory analyzer*. Domovské stránka tohoto projektu se aktuálně nachází na [5], kde je nástroj ke stažení a kde najdeme spoustu užitečných informací.

Nástroj pracuje pouze nad částí paměti – haldou (*heap*). Musíme si proto nechat vytvořit obraz haldy k nějakému okamžiku. Ten se uloží do souboru, nejčastěji s příponou hprof. Takový soubor označujeme jako ‚*heap dump*'. Máme-li *heap dump*, můžeme jej otevřít v MAT a nechat jej předzpracovat. MAT provede předdefinovanou analýzu haldy a vygeneruje nám základní přehled zjištěných informací – *Leak suspects report*. Mimo jiné připraví obraz haldy na další zpracování.

7.3.1.1. Leak Suspects report

Jednou z velice užitečných funkcí je zmíněný Leak suspects report. Provede základní analýzu haldy a na základě definovaných algoritmů se pokusí nalézt *memory leak* – jeden nebo více, pokud jsou. Ty mohou být viditelné jako mnoho instancí jednoho objektu (neuzavřená připojení apod.), nebo několik velkých objektů atd. S tím MAT počítá a implementované algoritmy projdou právě takové možnosti a výsledek provedené analýzy zobrazí jako report popisující možné problémy.

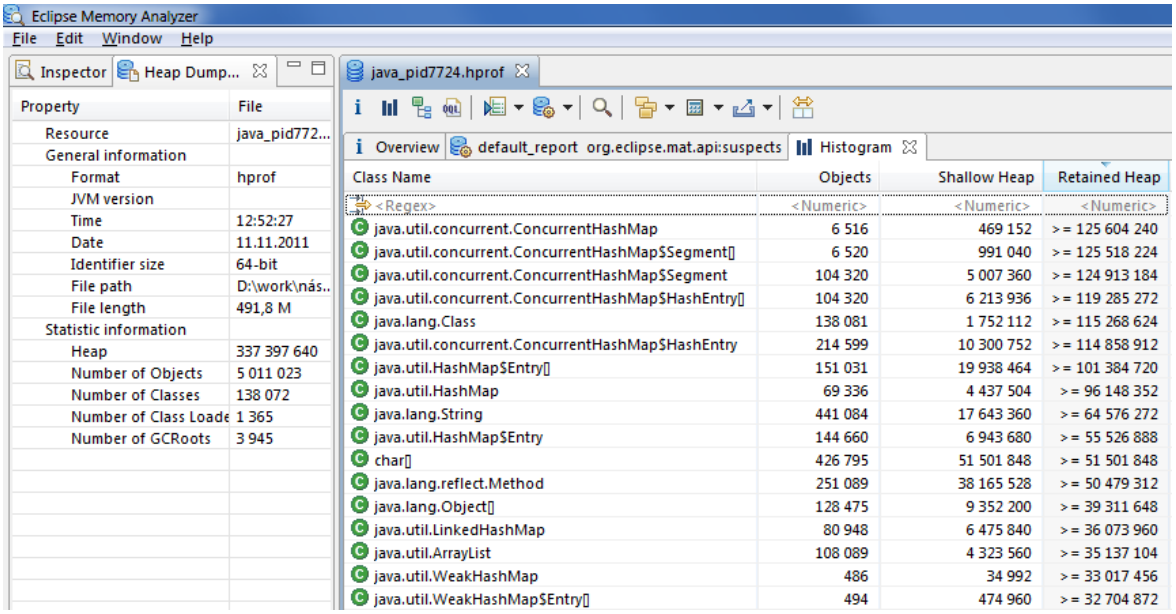
Občas se podaří, že hned z tohoto reportu zjistíme příčinu problému. Obvykle však na to spoléhat nemůžeme a musíme pokračovat v analyzování haldy.



Obrázek 23 - Leak suspects report

7.3.1.2. Histogram

Další užitečnou funkcí nástroje je Histogram. Ten nám poskytne informaci o četnosti instancí jednotlivých tříd a velikosti místa v paměti, které zabírají. Instance pak můžeme řadit podle jejich počtu, nebo podle zabírané velikosti. Na obrázku dále vidíme ukázku histogramu seřazeného podle velikosti, konkrétně podle *Retained heap size*, viz dále.



Obrázek 24 - Histogram

7.3.1.3. *Dominator tree*

Dalším užitečným pohledem na haldu je *dominator tree*. Ukáže nám, které objekty zabírají v haldě nejvíce místa. Na dalším obrázku je opět ukázka objektů seřazených dle velikosti, kterou zabírají. Dále je možné seřadit dle podílu (v procentech), jaký v haldě zabírají.

Overview | default_report org.eclipse.mat.api:suspects | Histogram | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class sun.awt.AppContext @ 0xb4b956f0 System Class	64	12 666 992	3,75%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xe93a5f60	368	7 414 688	2,20%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xe3a07d08	368	7 325 512	2,17%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xe605cef8	368	7 312 384	2,17%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xe17374a8	368	7 310 504	2,17%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xed2714f8	368	7 302 864	2,16%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xe859a680	368	7 302 280	2,16%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xeac705d0	368	7 277 976	2,16%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xe482fcb0	368	6 776 936	2,01%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xee06e9a0	368	6 755 184	2,00%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xe6eaba88	368	6 666 232	1,98%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xeba6b060	368	6 582 288	1,95%
weblogic.utils.classloaders.GenericClassLoader @ 0xef2eda00	152	5 130 608	1,52%
weblogic.descriptor.internal.MarshallerFactory @ 0xe0748a18	32	4 757 688	1,41%
class org.quartz.utils.DBConnectionManager @ 0xda994bb8	16	4 180 936	1,24%
class org.quartz.utils.DBConnectionManager @ 0xd4866310	16	3 942 528	1,17%
class org.quartz.utils.DBConnectionManager @ 0xce37aab8	16	3 933 856	1,17%

Obrázek 25 - Dominator tree

Jak již název napovídá, zobrazujeme si zde stromovou strukturu vazeb mezi jednotlivými objekty. Ukázka je na dalším obrázku. Zobrazovat je možné od kořene GC, nebo naopak od konkrétního objektu ke kořenu GC. Můžeme zde například zjistit, že problémem je třeba velká kolekce, kde po zobrazení stromu referencí nalezneme velké množství instancí jiných objektů, které by naši pozornost jinak neupoutaly, ale shluknuté v jedné kolekci již působí podezřele. Toto poněkud vágní, obecné vyjádření se nedá bez popisu konkrétní situace konkretizovat, protože je závislé právě na analyzované aplikaci. Z toho vyplývá, že se při řešení těchto problémů většinou neobejdeme bez znalosti konkrétní aplikace. Můžeme provézt analýzu i bez této znalosti, ale výstupem bude pouze nasměrování někoho se znalostí dané aplikace, kam by se měl podívat, co by měl prověřit.

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class sun.awt.AppContext @ 0xb4b956f0 System Class	64	12 666 992	3,75%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0xe93a5f60	368	7 414 688	2,20%
weblogic.servlet.internal.WebAppServletContext @ 0xe7d6e9f8	392	6 080 832	1,80%
weblogic.servlet.internal.AttributesMap @ 0xe7f0e9b8	56	4 794 528	1,42%
java.util.concurrent.ConcurrentHashMap @ 0xe7f0e9d8	72	4 794 472	1,42%
weblogic.servlet.internal.WebComponentContributor @ 0xe7f1abb8	72	888 832	0,26%
weblogic.servlet.internal.WebAppModule @ 0xe7d6eac8	192	120 008	0,04%
weblogic.application.internal.ApplicationContextImpl @ 0xe7d66f48	384	95 528	0,03%

Obrázek 26 - Dominator tree - rozbalený strom

Při prohlížení zobrazených dat můžeme kromě uvedeného seřazování používat také regulární výrazy, což pak cvičenějším analytikům pomůže k rychlejšímu dohledání možných příčin problémů.

7.3.1.4. OQL

Dalším velice užitečným nástrojem je možnost použití Objektového dotazovacího jazyka OQL (Object Query Language). Haldu si můžeme představit jako objektově orientovanou databázi, nad kterou provádíme dotazy. Tímto způsobem můžeme dostat pohledy na haldu, které nám předchozí nástroje neumožní, zobrazovat si jen část haldy atd.

Nejlépe možnosti tohoto nástroje ukážou příklady. Například chceme-li si zobrazit objekty tříd z balíku com.vsb.fei.*, napíšeme následující dotaz:

```
SELECT * FROM "com\.vsb\.fei\..*"
```

Občas nás může dokonce zajímat konkrétní hodnota některého objektu. Můžeme si tak zobrazit například všechny instance třídy java.lang.String, které obsahují text ‚*diploma*'. Dotaz pak bude vypadat následovně:

```
SELECT * FROM java.lang.String s
     WHERE toString(s) = "diploma"
```

Jak vidíme, možnosti jsou široké a záleží pak na zkušenostech, jak jich umíme využít ke zjištění problémů v našich aplikacích. Je však důležité mít na paměti, že nám uvedené nástroje pracují nad obrazem haldy (v nějakém okamžiku) a neposkytnou nám ani dynamický pohled na aplikaci, ani informace, které jsou někde mimo haldu.

K dynamickému pohledu na aplikaci nám poslouží daleko jiné způsoby, například analýza *thread dump*, tedy obrazu vláken aplikace v nějakém okamžiku. Vytvořením takovýchto obrazů například v minutových intervalech nám již dá dynamický pohled na aplikaci, například kde došlo k uvíznutí při čekání na zámek atd.

7.3.1.5. Některé použité pojmy

V textu je použito několik pojmů, které je nutné pochopit pro práci s tímto užitečným nástrojem. Následuje proto krátký popis těch nejdůležitějších.

- **Retained set** – sada objektů, které by byly sebrány garbage collectorem.
- **Retained size** – suma plytkých velikostí všech objektů v retained set. Retained size objektu je taková velikost paměti, která by byla uvolněna, pokud by GC sebral tento objekt.
- **Shallow size** – plytká velikost jednoho objektu. Jde o velikost daného objektu bez velikostí objektů z něj odkazovaných. Například shallow size instance `java.lang.String` nebude obsahovat množství paměti nutné pro základní `char[]`.

7.3.1.6. Nástin postupu při řešení problémů

Jak tedy uvedených znalostí využít při řešení konkrétních problémů? Postup může být následující:

- Necháme si vytvořit heap dump, nejlépe v okamžiku pádu aplikace na OutOfMemory výjimku. Toho docílíme buď přidáním parametru pro start Java procesu, nebo pomocí externích nástrojů (jmap, …).
- Spustíme Eclipse memory analyzer a načteme v něm hrpof soubor.
- Zobrazíme si *Leak Suspects report* a projdeme jeho výsledky. Pokud se v této fázi dozvíme informace odkazující na možný problém v aplikaci, projdeme zdrojový kód aplikace. Problém může být špatně napsaný kód, špatné použití některé externí knihovny, nebo také chyba v externí knihovně.
- Pokud možná příčina není zřejmá na první pohled, zobrazíme si histogram objektů v haldě. Je možné, že nalezneme velké množství neuzavřených *socket* připojení v podobě sady sice malých objektů, ale ve velkém množství. Také je možné, že zde nalezneme několik málo objektů, které zabírají převážnou část haldy. Takové si můžeme zobrazit pomocí *Dominator tree*.
- Pomocí dominator tree si opět můžeme seřadit objekty podle jejich velikosti. Největší konzumenty pak prověříme, odkud jsou odkazovány (směrem ke kořenu GC), nebo naopak jaké objekty odkazují. To je případ, kdy jejich mělká velikost nebude velká, ale retained size tvoří velkou část prostoru přiděleného haldě.
- Jakékoli zjištění většinou směřuje k analýze zdrojového kódu aplikace. Pokud se ale nacházíme v prostředí aplikačního serveru, na kterém běží několik aplikací, po takové analýze zaměříme naši pozornost na jednu konkrétní aplikaci a následně na konkrétní zdroje problémů.

Výše uvedené je pouze nástinem možného řešení a to nikoli dogmatickým. Řešení těchto problémů vyžaduje jednak znalosti o práci správce paměti dané implementace JVM, viz kapitoly 7.1 a 7.2, dále pak jisté zkušenosti s prací s nástroji typu MAT. V neposlední řadě je potřeba znát aplikaci, ve které se problém nachází a také použité aplikační rámce, jejichž nesprávné použití může být také zdrojem problémů.

8. Závěr

Hlavním cílem této práce bylo seznámit s tím, co nabízí technologie prezentované pod souhrnným označením Fusion Middleware od společnosti Oracle Corporation. Práce proto seznámila s rozsahem toho, co tento technologický balík nabízí a dokázala, že Oracle, ač dlouhá léta chápán především jako přední dodavatel řešení pro perzistenci dat, nyní nabízí opravdu komplexní řešení jakékoli infrastruktury, ať nově budované, nebo původní rozšiřované.

Dále se práce zaměřila na předem požadované části Fusion Middleware, kterými jsou Oracle Application Development Framework a Oracle WebLogic, přičemž tyto produkty představila a popsala jejich vlastnosti a možnosti. Při přípravě této části byly velice užitečné podklady pro přípravu na certifikaci WebLogic serveru. Jenou z požadovaných technologií byl také Spring framework, který byl představen, ale jelikož nakonec nebyl (z důvodů uvedených v části implementace) použit, nebyla mu věnována taková pozornost jako zbylým technologiím.

Cílem praktické části bylo osvojit si ne jen teoretické, ale také praktické znalosti vybraných Oracle technologií. Bylo proto potřeba nastudovat tvorbu aplikací při použití prostředí Oracle JDeveloper, se kterým jsem se do té doby nesetkal. Dále bylo potřeba osvojit si způsoby použití aplikačního rámce Oracle ADF. Díky způsobu, jakým se s tímto rámcem pracuje, se jednalo o poměrně zajímavou a novou práci, jelikož ubylo obvyklého programování a mnohé rutinní operace se prováděly za pomoci rámce rychle. Na druhou stranu je potřeba říci, že přibylo nastavování komponent, úprav již vygenerovaných částí apod. Z počátku tak vývojář vytvoří některé části bez úplného chápání souvislostí a ty pochopí až posléze. Ačkoli je to tedy velice intuitivní rámec, i s jeho použitím je potřeba mít jisté zkušenosti, chceme-li tvořit rozsáhlejší aplikace pro firemní zákazníky. Nevýhodou také může být, že i pro údržbu takovýchto aplikací je potřeba znát tuto technologii, což v praxi, alespoň zatím, může zužovat výběr kandidátů na tyto pozice, neboť znalost Oracle ADF je jistě méně rozšířená, než znalost platformy Java. Obecně se však dá říci, že se jedná o mocný nástroj a ukazuje trend, jakým se ubírá vývoj těchto technologií.

Jelikož se představené technologie týkají platformy Java, která je koneckonců sama chápána jako součást balíku Fusion Middlevare, byla představena také problematika správy paměti a nastíněn možný způsob řešení problému známého jako *memory leak*. Hlavním důvodem bylo, že je to o poměrně častý problém u J2EE aplikací a ne jinak je tomu u aplikace postavené na ADF a běžící na WebLogic Serveru, ať už jako samostatná webová aplikace, nebo jako WSRP producent v portálové aplikaci postavené na WebLogic Portal.

9. Literatura

[1] Johnson, R. a Hoeller, J.: Expert One-on-one: J2EE Development Without EJB, Willey Publishing, Inc., 2004, 0-7645-5831-5

[2] Jmap - Memory Map. *Java SE Documentation* [online]. 2011, [cit. 2012-03-30]. Dostupné z: http://docs.oracle.com/javase/6/docs/technotes/tools/share/jmap.html

[3] PermGenSpace problem? No problem!. *PermGen problem* [online]. 2011, [cit. 2012-02-30]. Dostupné z WWW: http://blog.novoj.net/2008/04/11/permgenspace-problem-no-problem.

[4] THON, Ladislav. Správa paměti na platformě Java. *Správa paměti na platformě Java* [online]. 2011, s. 28 [cit. 2012-03-30]. Dostupné z WWW: http://cw.zcu.cz/wps/PA_Courseware/DownloadDokumentu?id=3692

[5] Eclipse : Memory analyzer [online]. 2011 [cit. 2011-11-26]. Dostupné z WWW: http://www.eclipse.org/mat.

[6] SENSHARMA, Jay. How to Generate HeapDump in case of Windows Service. *Java Heap*: *How to Generate HeapDump in case of Windows Service* [online]. 2011, [cit. 2012-01-30]. Dostupné z WWW: http://middlewaremagic.com/weblogic/?tag=java-heap.

[7] Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine. *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine* [online]. 2003, [cit. 2012-01-30]. Dostupné z WWW: www.oracle.com/technetwork/java/gc-tuning-5-138395.html.

[8] BEA WebLogic Server 9.0: J2EE : Programing with Servlets and JSPs in Eclipse. San Jose, CA 95131 : BEA Systems Inc., 2005. 618 s.

[9] Oracle *:* Oracle Application Development Framework [online]. 2011 [cit. 2011-11-26]. Dostupné z WWW: http://www.oracle.com/technetwork/developer-tools/adf/overview/index.html.

[10] Oracle : Oracle Fusion Middleware [online]. 2011 [cit. 2011-11-26]. Dostupné z WWW: http://www.oracle.com/cz/products/middleware/index.html.

[11] PECINOVSKÝ, Rudolf. Návrhové vzory, CPress, Brno 2007, 978-80-251-1582-4

[12] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Design paterns. Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995. 396 s. ISBN 0-201-30998-X

[13] J2EE Architecture. Oracle Weblogic [online]. 2012, [cit. 2012-03-30]. Dostupné z WWW: <http://middleware-weblogic.blogspot.com/p/j2ee-architecture.html>.

[14] Understanding Domain Configuration. *Oracle Technology Network* [online]. 2012, [cit. 2012-03-30]. Dostupné z WWW: http://docs.oracle.com/cd/E15051_01/wls/docs103/domain_config/understand_domains.html

[15] SOA. *Service-Oriented Architecture* [online]. 2012, [cit. 2012-03-30]. Dostupné z WWW: http://kore.fi.muni.cz/wiki/index.php/SOA.

[16] PICHLÍK, Roman. Spring Framework: představení J2EE lightweight kontejneru. *Interval* [online]. 2012, [cit. 2012-03-30]. Dostupné z: http://interval.cz/clanky/spring-framework-predstaveni-j2ee-lightweight-kontejneru

[17] WLS tuning. *WebLogic* [online]. 2012, č. 111 [cit. 2012-05-1]. Dostupné z: http://docs.oracle.com/cd/E17904_01/web.1111/e13814/wls_tuning.htm#i1148994

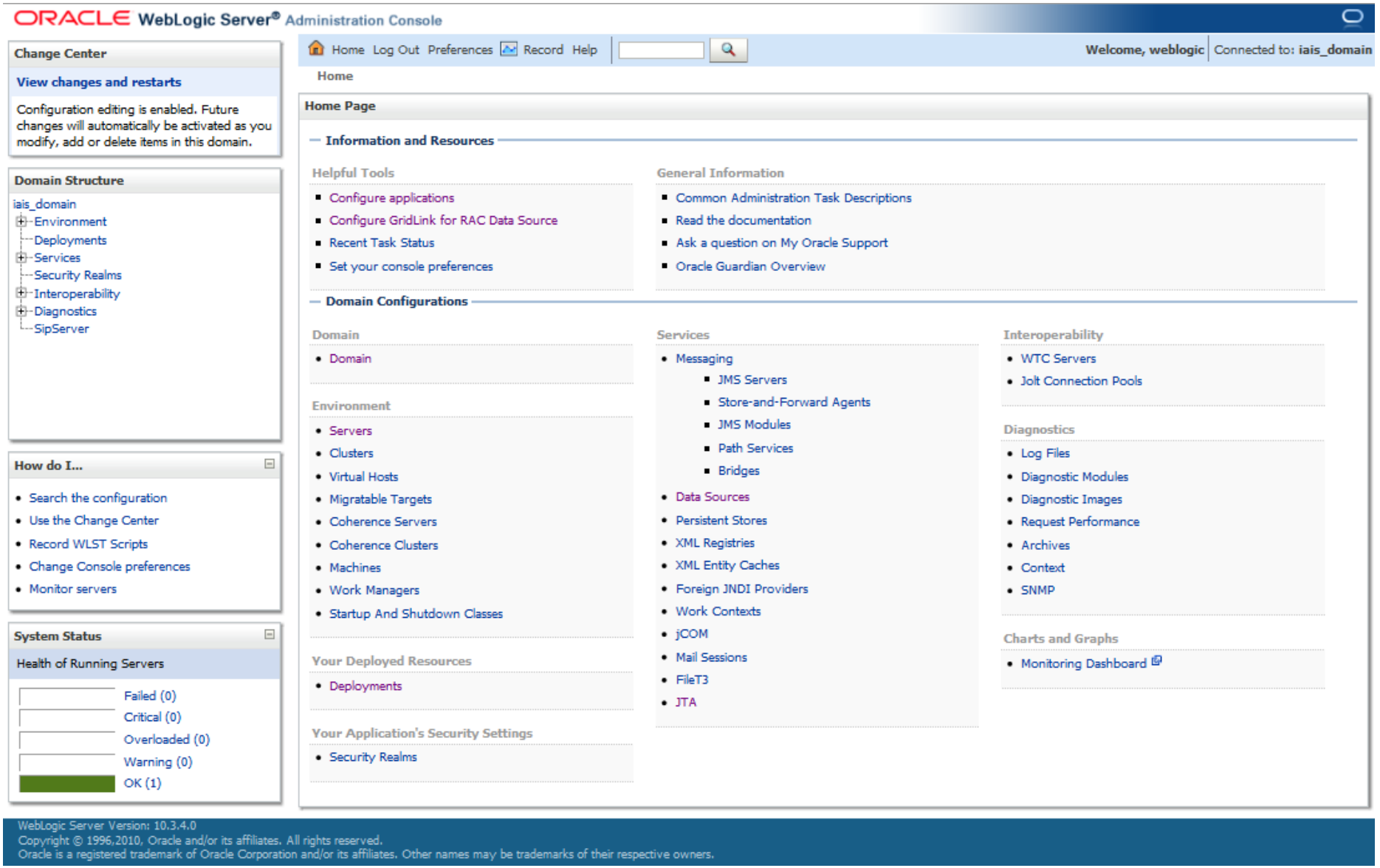
10. Přílohy

10.1 CD

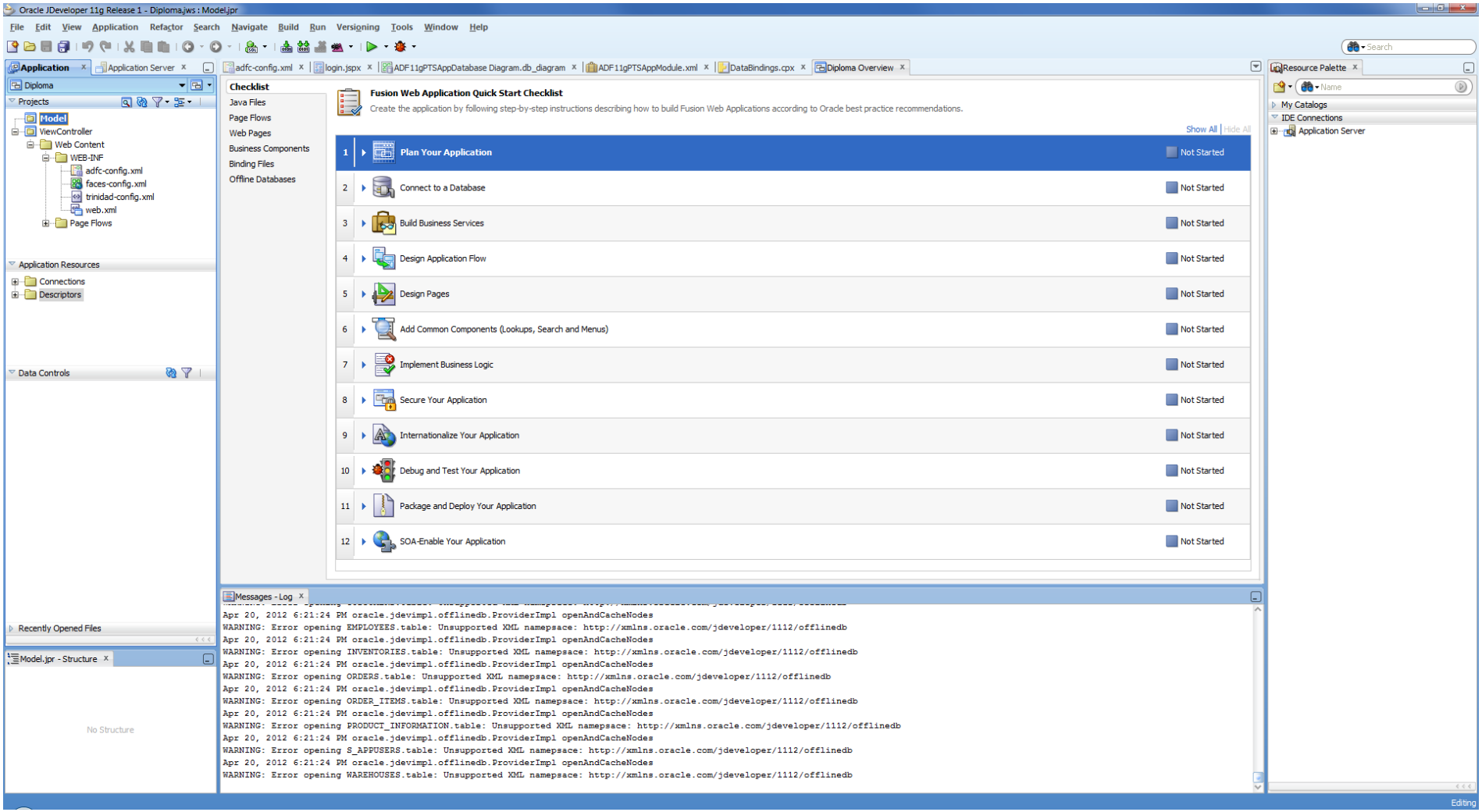
- tento dokument — kořenový adresář
- zdrojové soubory — ./DP_1

10.2 Schémata a obrázky

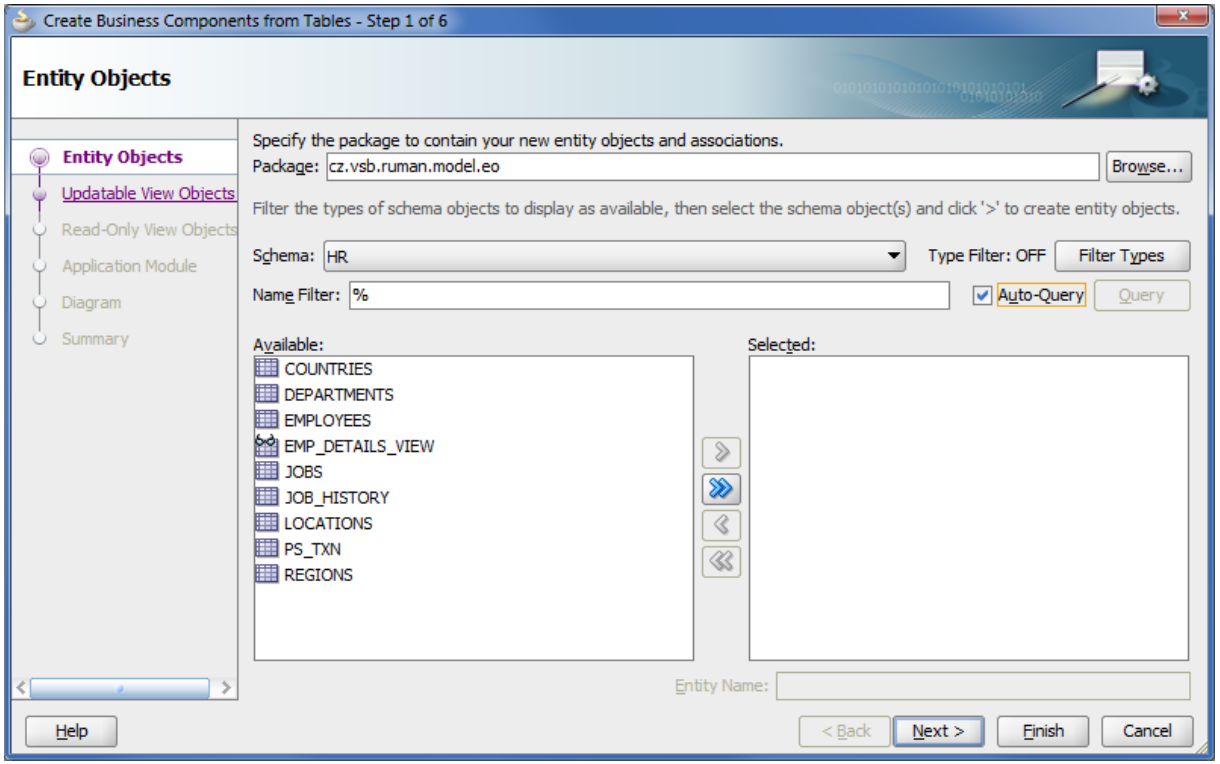
Příloha A – Oracle WebLogic Administration Console
Příloha B – Oracle JDeveloper 11g
Příloha C – Průvodce vytvořením business komponent
Příloha D – Metoda login action



Příloha A – Oracle WebLogic Administration Console



Příloha B – Oracle JDeveloper 11g (v 11.1.1.0)



Příloha C – Průvodce vytvořením business komponent

```
public String cmdLogin_action() {
    BindingContainer bindings = getBindings();
    OperationBinding operationBinding = bindings.getOperationBinding("checkUser");
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        return null;
    }
    int ret = ((Integer)result).intValue(); // convert the result of checkUser into int type
    if (ret == 0) {
        String message = "Invalid login";
        FacesContext.getCurrentInstance().addMessage(null,
                                                     new FacesMessage(message)); // add a message in the faces conte
        return null; // the same login page is still displayed
    } else {
        String userId =
            bindings.getControlBinding("userid").toString(); // read the value from the bindings for the userid text
        System.out.println("userId = " + userId);
        HttpServletRequest req =
            (HttpServletRequest)FacesContext.getCurrentInstance().getExternalContext().getRequest();
        req.getSession().setAttribute("login",
                                      userId); // write the value of the Userid in a session attribute named "login"
        return "success"; // navigation to the page pointed by the Navigation Case "success"
    }
}
```

Příloha D – metoda login action